assignement2

Lilach Herzog & Leslie Cohen

13 5 2022

Our whole project and code can be found on github at https://github.com/LeslieLebon/assignment-2.git

Our objective in this project was to analyze the "wheat-seeds" dataset (where the 'Type' of the seed is the column used for labeling and classifying) using two classifier models that we built. We decided on using the KNN algorithm and the Decision Tree (DT) algorithm.

1 preproccessing

1.1 Upload data and libraries

```
library(gmodels)
library(C50)
library(class)
library(tidymodels) # for the rsample package, along with the rest of tidymodels
library(ROCR)
library(reshape2)
library(ggpmisc)
library(ggplot2)
library(gridExtra)
library(pROC)

seeds <- read.csv("seeds.csv")# read data
set.seed(1234)#to initialize a pseudo random number generator.
```

1.2 split into training and test sets

1.2.1 initial look at the data

We know the class 'Type' is the column (label) to classify according to. We took a quick look at the data as a whole, and specifically the 'Type' column. In order to further understand the data and be able to work with it we also used the "table" function. this function builds a contingency table of the counts at each combination of factor levels- we used it for the 'Type'.

```
str(seeds) #quick look at the data as a whole
```

```
## 'data.frame':    199 obs. of  8 variables:
##  $ Area           : num  15.3 14.9 14.3 13.8 16.1 ...
##  $ Perimeter      : num  14.8 14.6 14.1 13.9 15 ...
##  $ Compactness    : num  0.871 0.881 0.905 0.895 0.903 ...
##  $ Kernel.Length  : num  5.76 5.55 5.29 5.32 5.66 ...
##  $ Kernel.Width   : num  3.31 3.33 3.34 3.38 3.56 ...
##  $ Asymmetry.Coeff: num  2.22 1.02 2.7 2.26 1.35 ...
##  $ Kernel.Groove  : num  5.22 4.96 4.83 4.8 5.17 ...
```

```
##  $ Type           : int  1 1 1 1 1 1 1 1 1 1 ...
```

```
table(seeds$Type) # quick look specifically at the 'Type' column
```

```
##
##  1  2  3
## 66 68 65
```

The column we want to classify is 'Type'. There are 3 Types of seeds found in in the column 'Type': 1, 2 and 3 (We will sometimes refer to them as Type_1, Type_2 and Type_3, to make it easier to understand and differentiate them from numbers that represent other things). We can see the Type has 66 times Type_1, 68 times Type_2, and 65 times Type_3, about a third of each Type.

Before starting on teaching the algorithm, we converted the 'Type' column to a factor, as that is what is required by the C50 package.

```
seeds$Type<-as.factor(seeds$Type)
```

Since the whole table is sorted according to the 'Type' column, and we want to work randomly, we will shuffle the whole table

1.2.2 Splitting

```
length_of_training_set <-round(0.8*(nrow(seeds)),0)
train_set <- seeds[1:length_of_training_set, ]
test_set <- seeds[(length_of_training_set+1):nrow(seeds), ]
train_set_Types <-train_set[, 8]
test_set_Types <-test_set[, 8]
prop.table(table(train_set$Type))
```

```
##
##         1         2         3
## 0.3081761 0.3333333 0.3584906
```

Since we know that there are about a third of each Type in the data set, we wanted to be sure that the distribution is as we defined (about 1/3 of each Type in of both training and test sets)

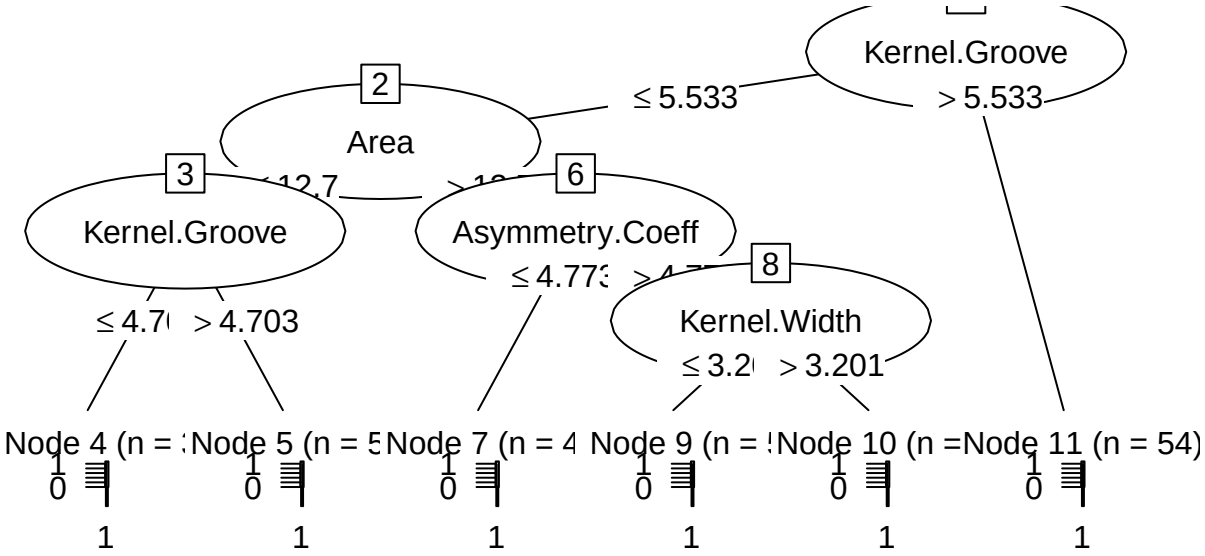
2 Decision Tree

A Decision Tree uses a series of questions that lead to a decision- a class. The goal is to improve purity, to try and have a higher and higher percentage of items in the node are similar, until all items in a node belong to the same class. It does so by taking the items in a node and splitting them in such a manner that they become two groups with improved purity in each (more samples in the group belong to the same class).

2.1 initial run

We will use the C5.0 method. The 8th column of the dataset is the Type class variable, so we need to exclude it from the training data frame, and supply it as the target factor (label) vector for classification:

```
DT_model <- C5.0(train_set[-8], train_set$Type)
plot(DT_model)
```



2.1.1 Explanation of initial tree:

```
summary(DT_model)
```

The algorithm did six iterations. In the first iteration it divided all the seeds into two subgroups (nodes) according to whether their Groove was higher or lower than 5.533. One node was pure (all seeds in the node were Type_2), and the other node went through another iteration according to the Area, where both nodes were not pure and needed to go through another iteration. One node was split into two pure nodes-again according to Groove-which yielded 2 pure nodes (Type_3 or Type_1). The other node was split according to the asymmetry coefficient which resulted in one pure node (of Type_1) whereas the other node went into another iteration according to the width which yielded 2 pure nodes (types 3 and 1).

The classifier made 3 mistake for an error rate of 1.9%, which is relatively low.

After teaching the model on the training set, we want to use it to predict the class is on the test set, and then compare the test set to what we predicted for the test said using our predictive model.

```
DT_pred <- predict(DT_model, test_set)
DT_table<-CrossTable(test_set$Type, DT_pred, prop.chisq = FALSE, prop.c = FALSE,
                     prop.r = FALSE, dnn = c('actual Type', 'predicted Type'))
DT_first_tbl<-DT_table[["t"]]
```

```
##    y
## x    1  2  3
##   1 17  0  0
##   2  1 14  0
##   3  2  0  6
```

Type seeds 2 and 3 both have no false positives (all predictions of Type_2 or three were actually Types 2 or 3) with a few false negatives (one Type_2 and 2 Type_3s that were predicted as Type one). Type_1 on the other hand has no false negative (all Type_1 seeds were predicted as Type_1) but it had a few false positives (as mentioned before- one that was actually Type_2 and two were actually Type_3).

We computed the accuracy test from the confusion matrix (the proportion of true positive and true negative over the sum of the matrix).

```
acc_DT_Test <- sum(diag(DT_table[["t"]])) / sum(DT_table[["t"]])
```

Accuracy of 92.5% is pretty good, but we will try to improve our results using adaptive boosting (a process

in which many Decision Trees are built and the trees vote on the best class for each example).

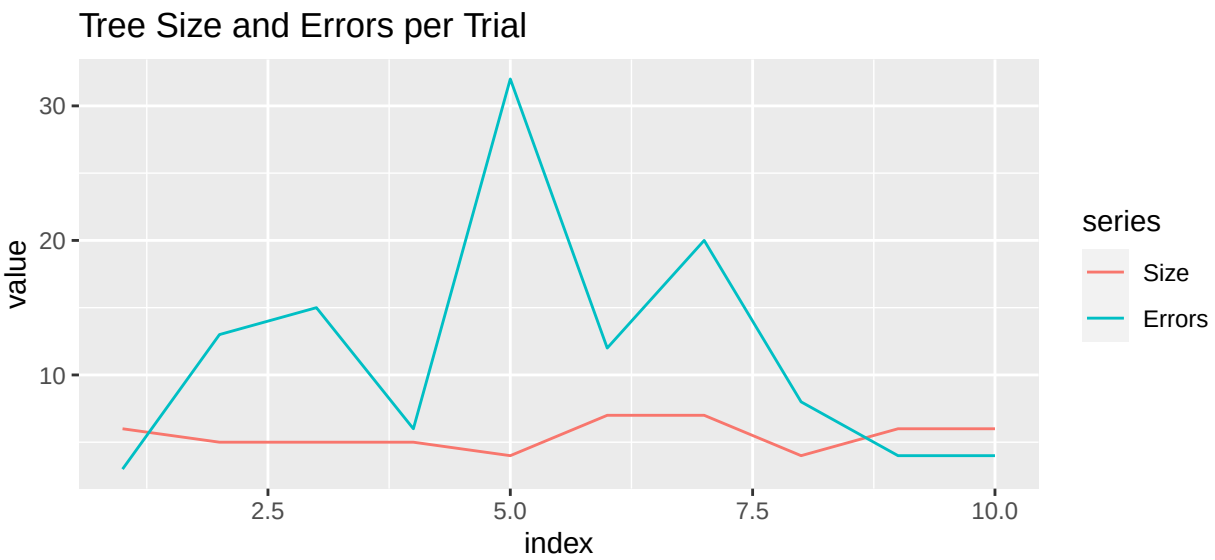
2.2 Adaptive Boosting

Adaptive boosting is a process in which many Decision Trees are built and the trees vote on the best class for each example.

2.2.1 10 trials

We'll start with 10 trials, a number that has become the de facto standard, as research suggests that this reduces error rates on test data by about 25 percent:

```
DT_boost10 <- C5.0(train_set[-8], train_set$Type, trials = 10)
summary(DT_boost10 )
DT_boost10_tbl<-DT_boost10[["boostResults"]]
df10 <- data.frame(index=c(DT_boost10_tbl$Trial),Size=c(DT_boost10_tbl$Size),Errors=c(DT_boost10_tbl$Er
df10_melted <- melt(df10 ,  id.vars = 'index', variable.name = 'series') #melt data frame into long for
ggplot(df10_melted, aes(index, value)) + geom_line(aes(colour = series)) +labs(title = "Tree Size and E
```



The classifier made 1 mistake for an error rate of 0.6% percent. This is an improvement over the previous training error rate. However, it remains to be seen whether we see a similar improvement on the test data. Let's take a look at how good the prediction is:

```
DT_boost10_pred <- predict(DT_boost10, test_set)
DT_table_10<-CrossTable(test_set$Type, DT_boost10_pred,
prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
dnn = c('actual Type', 'predicted Type'))
DT_10_tbl<-DT_table_10[["t"]]
acc_DT_Test10 <- sum(diag(DT_10_tbl)) / sum(DT_10_tbl)
```

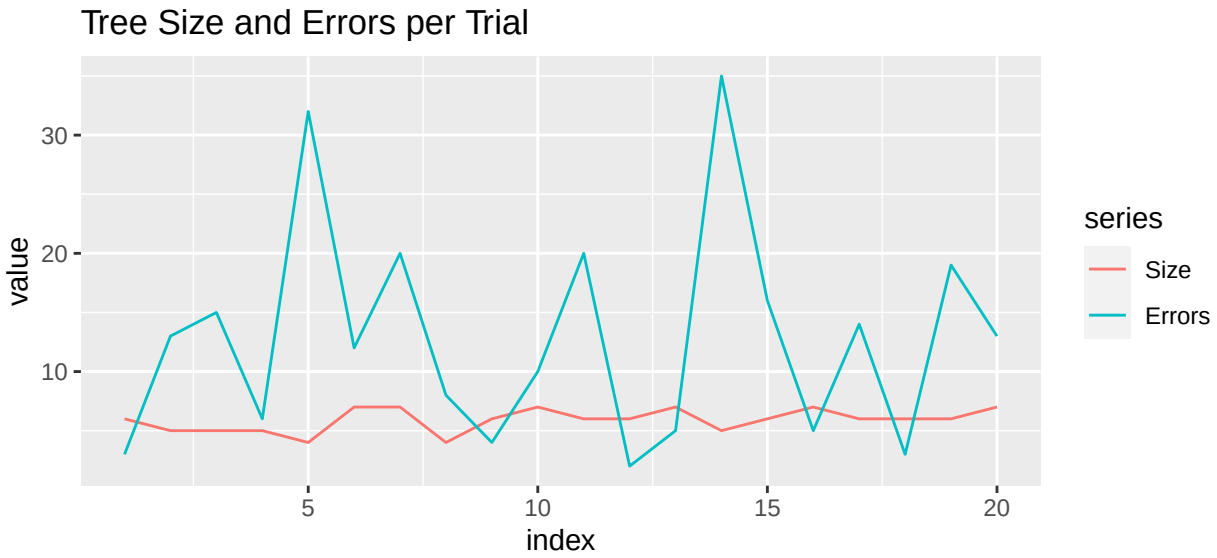
```
##    y
## x    1  2  3
##   1 17  0  0
##   2  1 14  0
##   3  1  0  7
```

The accuracy improved from 92.5% to 95%: it made 1 false negative (predicted Type_1) in Type_3 instead of 2 (which also lowered the false positive of Type_1 to 2 instead of 3) before the boost.

2.2.2 20 trials

WE will try to improve the result, with more trials than 10, for example 20.

```
DT_boost20 <- C5.0(train_set[-8], train_set$Type, trials = 20)
DT_boost20_trials<-DT_boost20[["boostResults"]]
DT_boost20_trials
summary(DT_boost20 )
```



The classifier with 20 trials made 0 mistake for an error rate of 0.0% percent(!).

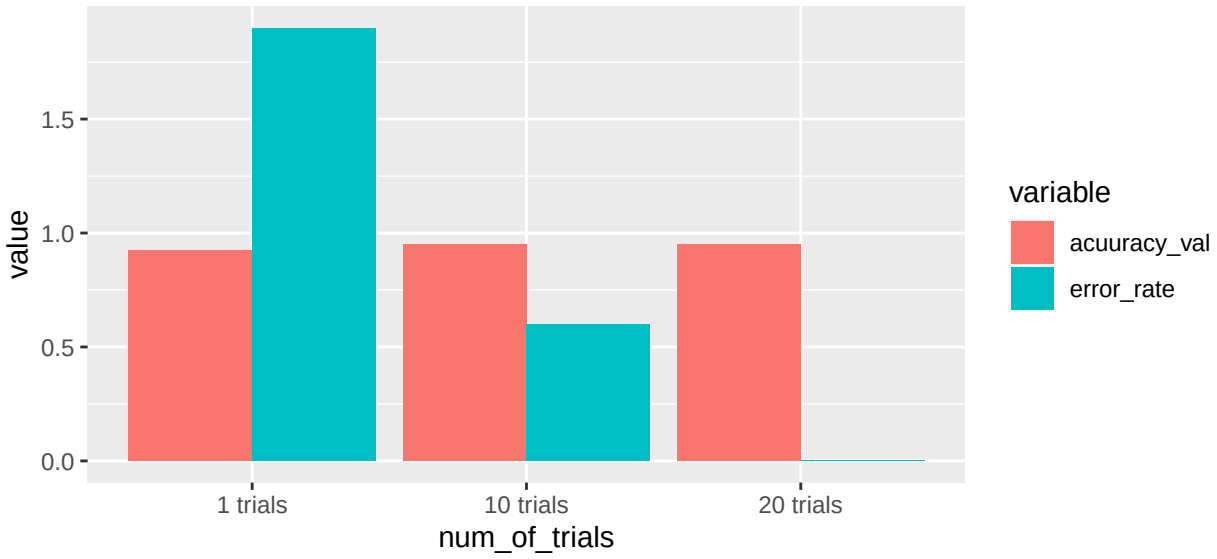
```
DT_boost20_pred <- predict(DT_boost20, test_set)
DT_table_20<-CrossTable(test_set$Type, DT_boost20_pred,
prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
dnn = c('actual Type', 'predicted Type'))
DT_20_tbl<-DT_table_20[["t"]]
acc_DT_Test20 <- sum(diag(DT_20_tbl)) / sum(DT_20_tbl)
```

```
##    y
## x    1  2  3
##   1 17  0  0
##   2  1 14  0
##   3  1  0  7
```

The training was better, but the accuracy stayed exactly the same (95%) Indeed, choosing 20 trials in the boost does not improve the result. The error rate stay similar than with 10 trials, so there is no reason to do 20 trials.

2.3 DT conclusion

```
acuuracy <- data.frame(num_of_trials=c("1 trials","10 trials","20 trials"),
                       acuuracy_val=c(acc_DT_Test,acc_DT_Test10,acc_DT_Test20),
                       error_rate=c(1.9, 0.6, 0.0))
acuuracy_melted <- melt(acuuracy)
both<-ggplot(data = acuuracy_melted, aes(num_of_trials,value , fill=variable)) +
  geom_col(position = 'dodge')
grid.arrange(both+ theme(legend.position="right"), ncol=1, nrow =1)
```



The Decision Tree algorithm gave us accuracy of 92.5% to 95% .we increased the accuracy by boosting it with 10 or 20 trials. Our dataset is relatively small so we didn't see a change in accuracy between 10 trials and 20 trials, but because the error rate in the 20 trial run managed to get to 0% as opposed to 0.6%, we assume that with a bigger dataset there will be a bigger difference between the amount of runs but in our case there is no difference so 10 trials is enough.

In addition to the classification itself, we were able to look at the process and the tree itself and learn some information about our data.

These is the attribute usages :

feature	first run	10 trials	20 trials
Kernel.Groove	100.00%	100.00%	100.00%
Area	66.04%	66.04%	100.00%
Asymmetry.Coeff	31.45%	100.00%	100.00%
Kernel.Width	4.40%	100.00%	100.00%
Perimeter	0.00%	100.00%	100.00%
Compactness	0.00%	0.00%	27.67%
Kernel.Length	0.00%	0.00%	18.87%

The algorithm uses the feature that provides the most Information Gain in every split. Since it used 100.00% of the information from the Kernel.Groove column in all runs, we can assume that that column provided the most information gain (at least in the first iteration). The algorithm also used the Area column, the Asymmetry.Coeff and the Kernel.Width columns in all 3 runs, so that also gives us some information about how much these features influenced the Information Gain function.

3 KNN Algorithm

In the KNN Algorithm, k is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the k training samples nearest to that query point.

3.1 run the KNN algorithm

And finally, we are ready to run our algorithm. We'll start with K of 3, the the number of Types we have and also an odd number, reducing the change of a tie vote.

```
pred_knn <- knn(train = train_set, test = test_set, cl = train_set_Types, k=3)
knn_table<-CrossTable(x = test_set_Types, y = pred_knn, prop.chisq=FALSE)
acc_knn3 <- sum(diag(knn_table[["t"]])) / sum(knn_table[["t"]])
df<-data.frame(predictions=c(as.numeric(pred_knn)),labels = c(test_set_Types))
pROC_knn3 <- roc(df$labels,df$predictions,
            smoothed = TRUE,
            ci=TRUE, ci.alpha=0.9, stratified=FALSE,
            plot=FALSE, auc.polygon=TRUE, max.auc.polygon=TRUE, grid=TRUE,
            print.auc=TRUE, show.thres=TRUE)
sens.ci <- ci.se(pROC_knn3)
```

Since the knn function returns a factor vector of the predicted values, we'll compare that vector with the true labels we saved in advance. We'll do the comparison with the CrossTable function, from the gmodels package loaded:

```
knn_table<-CrossTable(x = test_set_Types, y = pred_knn, prop.chisq=FALSE)
```

```
##    y
## x    1  2  3
##   1 17  0  0
##   2  0 15  0
##   3  0  0  8
```

We got an accuracy of 100%! We managed to classify our test set perfectly, now we want to see how other parameters will affect our perfect score

3.1.1 normalization

We tried normalization, thinking that the range of values of each parameter is similar in order to compare the distances of different features with different scales.

```
normalize <- function(x) {return ((x - min(x)) / (max(x) - min(x)))}
normalize(c(1, 2, 3, 4, 5)) # test our function 1
normalize(c(10, 20, 30, 40, 50)) # test our function 2
train_set_norm <- as.data.frame(lapply(train_set[1:7], normalize))
test_set_norm <- as.data.frame(lapply(test_set[1:7], normalize))
summary(train_set_norm$Area) # test our dataset
```

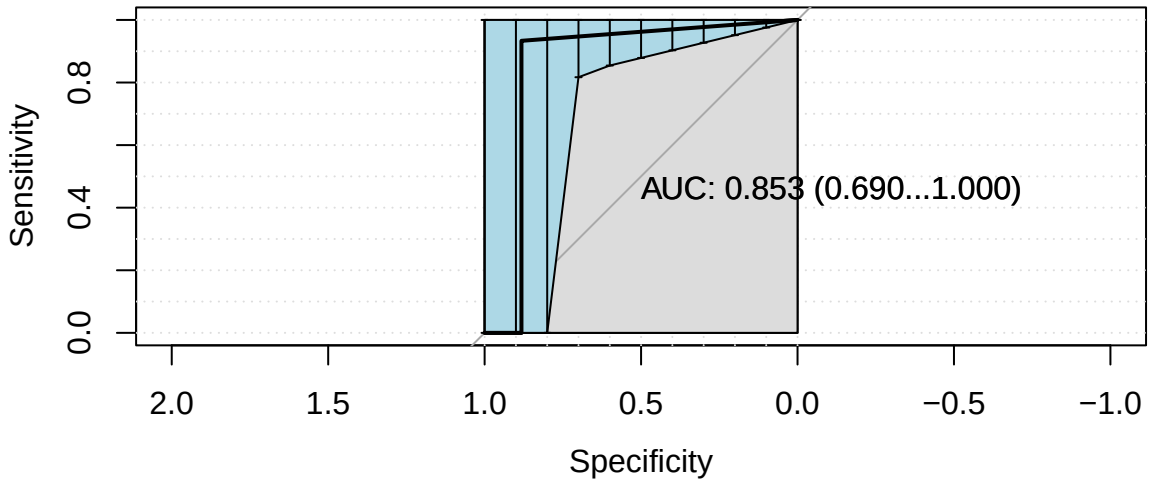
Everything looked ok so we went on the the prediction and evaluation of accuracy

```
pred_knn_norm <- knn(train = train_set_norm, test = test_set_norm, cl = train_set_Types, k=3)
knn_norm_table<-CrossTable(x = test_set_Types, y = pred_knn_norm, prop.chisq=FALSE)
acc_knn3_norm <- sum(diag(knn_norm_table[["t"]])) / sum(knn_norm_table[["t"]])
df_norm<-data.frame(predictions=c(as.numeric(pred_knn_norm)),labels = c(test_set_Types))
pROC_norm <- roc(df_norm$labels,df_norm$predictions,
            smoothed = TRUE,
            ci=TRUE, ci.alpha=0.9, stratified=FALSE,
            plot=FALSE, auc.polygon=TRUE, max.auc.polygon=TRUE, grid=TRUE,
            print.auc=TRUE, show.thres=TRUE)
sens.ci <- ci.se(pROC_norm)
```

```
##    y
## x    1  2  3
##   1 17  0  0
##   2  1 14  0
##   3  2  0  6
```

It looks like the normalization actually resulted in decreased accuracy, we plummeted from 1 to 0.925, as well as AUC that decreased from 1 to 0.967... In the above table we see that all 17 seeds(100%) of type 1 were predicted correctly, but it has a few false positives (3 seeds that were wrongly predicted as Type one: one seed was actually type 2 and 2 were type 3). on the other hand- types two and three which had no false positives (all seeds that were predicted as two or three were actually two or 3 respectively) and only a few false negatives (14/15 type 2 seeds (93.3%) and 6/8 type 3 seeds (75%) were identified correctly). Let's check z-score ### z-score standardization We also tried z-score standardization instead of normalization:

```
train_set_z <- as.data.frame(scale(train_set[1:7]))
test_set_z <- as.data.frame(scale(test_set[1:7]))
pred_z <- knn(train = train_set_z, test = test_set_z, cl = train_set_Types, k=3)
knn_z_table<-CrossTable(x = test_set_Types, y = pred_z, prop.chisq=FALSE)
acc_knn3_z <- sum(diag(knn_z_table[["t"]])) / sum(knn_z_table[["t"]])
df_z<-data.frame(predictions=c(as.numeric(pred_z)),labels = c(test_set_Types))
pROC_z <- roc(df_z$labels,df_z$predictions,
              smoothed = TRUE,
              ci=TRUE, ci.alpha=0.9, stratified=FALSE,
              plot=TRUE, auc.polygon=TRUE, max.auc.polygon=TRUE, grid=TRUE,
              print.auc=TRUE, show.thres=TRUE)
sens.ci <- ci.se(pROC_z)
plot(sens.ci, type="shape", col="lightblue")
plot(sens.ci, type="bars")
```



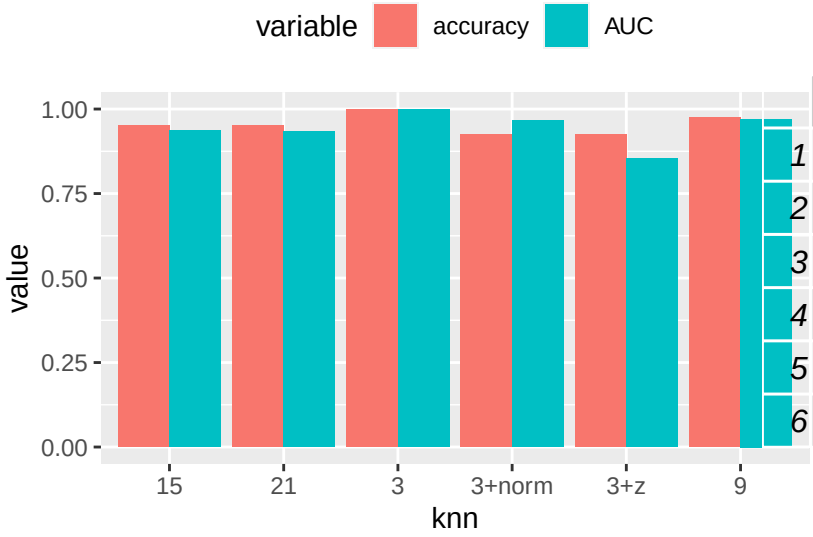

```
##    y
## x    1  2  3
##   1 15  0  2
##   2  1 14  0
##   3  0  0  8
```

We have the same decreased accuracy of 92.5% as with thhe normalized, but the AUC plummeted from 0.967 to 0.853. We can also see that with the normalization the 2 and 3 types had 100% TP and Type_1 had 100% TN, whereas with the z-score only Type_2 has 100% TP and Type_3 100% TN, whereas Type_1 now has 2/15 (11.8%) FN and 1/16 (6.7%) FP, so even though in both cases we had only 3 mistakes, in the normalization we had a problem just with the over-predictions of 1 while the rest was perfect, but with the z-score we have more problems..

3.1.2 different Ks

We tried with different K's (3, 9 and 15)

```
df_acc_auc<-data_frame(knn=c(3,9,15,21,"3+norm","3+z"), accuracy=c(acc_knn3,acc_knn9,acc_knn15,acc_knn2
                       AUC=c(pROC_knn3[["auc"]],pROC_knn9[["auc"]],pROCdf_knn15[["auc"]],
                             pROCdf_knn21[["auc"]], pROC_norm[["auc"]], pROC_z[["auc"]]))
df_acc_auc_melt<-melt(df_acc_auc)
plotted<-ggplot(data = df_acc_auc_melt, aes(knn,value, fill=variable)) +
  geom_col(position ='dodge')
table_norm<-tableGrob(df_acc_auc)
grid.arrange(plotted+theme(legend.position = "top"), table_norm,ncol=2,nrow=1, widths=c(2,1))
```



	knn	accuracy	AUC
1	3	1.000	1.0000(
2	9	0.975	0.97058
3	15	0.950	0.93725
4	21	0.950	0.93333
5	3+norm	0.925	0.96666
6	3+z	0.925	0.85294

We tried several more times with different K's (9 , 15 and 21) and it looks like the results are pretty similar with AUC of 0.853 for all 3 options.

With K = 9 we got one FP for Type_2 (it was actually Type_1) and an accuracy of 97.5%. With K = 15 we got in addition to the FP in type two another FP in type 1 (was actually type 2), and the accuracy was 95%. With K = 21 we don't have any FP in type two, but 2 FP in type 1 (was actually type 2), and the accuracy was also 95% (both 15 and 21 KNNs had 2 mistakes).

3.1.3 feature selection

The DT algorithm uses the feature that provides the most Information Gain in every split. Using that logic, the feature that is used the least provides the least information gain, and if it doesn't add much information it might disturb as noise. Thus, since we know that the accuracy of the k-NN algorithm can be severely degraded by the presence of noisy or irrelevant features, we tried the KNN algorithm once more, with K=15 (it might be easier to improve k=15 with 95% than k=9 with 97.5%) and without the "Kernel.Length" column (which attributed the least to the building of the DT).

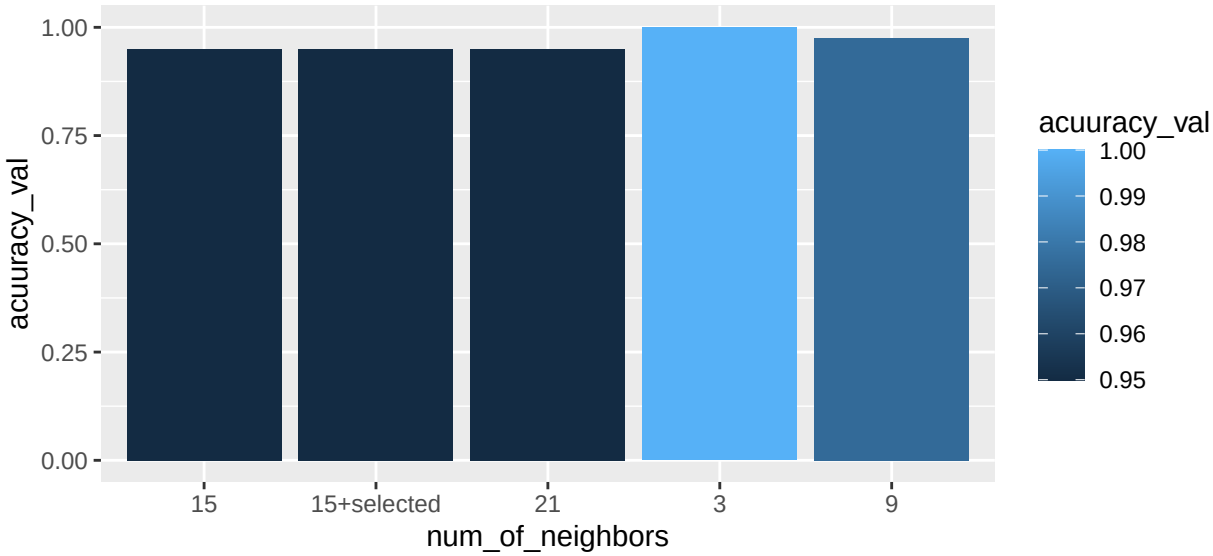
```
feat_selec<- knn(train = train_set[-4], test = test_set[-4], cl = train_set_Types, k=15)
knn9_no4_table<-CrossTable(x = test_set_Types, y = feat_selec, prop.chisq=FALSE)
acc_knn9_no4 <- sum(diag(knn9_no4_table[["t"]])) / sum(knn9_no4_table[["t"]])
```

There was no change on accuracy. We tried again without Compactness and again without both Compactness and Kernel.Length, and we still got to 97.5% accuracy

3.2 knn conclusions

```
knn_acuuracy <- data.frame(num_of_neighbors=c(3,9,15,21,"15+selected"),
                       acuuracy_val=c(acc_knn3,acc_knn9,acc_knn15,acc_knn21, acc_knn9_no4))
```

```
acuuracy_melted <- melt(knn_acuuracy)
ggplot(data = knn_acuuracy, aes(num_of_neighbors,acuuracy_val , fill=acuuracy_val)) + geom_col()
```



We managed to classify our data using the KNN algorithm, with k=3 (not normalized or standardized) we got 100% accuracy (and AUC of 1 of course). Our results show that- in general- the higher the k is the less accurate the algorithm is. We tried using K=9 and without the "Kernel.Length" (or "Compactness") column, but that didn't help the accuracy at all.

4 Conclusions

The Decision Tree algorithm worked well enough, with accuracy increased (using boosting) from 92.5% to 95%. The KNN algorithm was even better. We ran the KNN algorithm with four different Ks, and found that we get the best results using K = 3. Using the information from the Decision Tree algorithm we tried to increase the accuracy of our KNN algorithm, but it didn't help.