Final Project

Lilach Herzog & Leslie Cohen

6/10/2022

Upload data and libraries

```
glaucoma <- read.csv("GlaucomaM.csv")# read data
```

We have data from 196 patients, with 63 features. Since the whole table is already shuffled according to the 'Class' column, we don't need to shuffle the whole table again In order to get consistent results we initialized a pseudo random number generator. We also made sure to shuffle our data.

preproccessing

We want to classify our data according to the 'Class' column, which contains the information about whether all the information about a person (represented in a row) belongs to a healthy (normal) or sick (glaucoma) patient, divided neatly in half. Before starting on teaching the algorithm, we converted the 'Class' column to a factor.

```
glaucoma$Class<-as.factor(glaucoma$Class)
glaucoma_num_class<-as.numeric(glaucoma$Class)
```

EDA analysis

Results obtained with corrplot: Positive correlations are displayed in a blue scale while negative correlations are displayed in a red scale. we can observe 8 clusters of features that highly correlate. Results obtained with heatmap: we can observe one big cluster that could be divided in 2 and 2 others clusters of features.

We would like to observe if we can cluster patients healthy or with glaucoma with all the features. We performed first a PCA on all features and values.

Preform PCA on our data

```
default_pca <- prcomp(glaucoma[,1:62], center = TRUE,  scale. = TRUE)
summary(default_pca)
default_pca_plot <- autoplot(default_pca, data = glaucoma, colour = 'Class')
fit_pca <- lm(Class ~ .,
              cbind(Class = glaucoma[, "Class"], default_pca$x[, 1:2]) %>% as.data.frame())
```

In the PCA, we can observe 2 clusters of patients: glaucoma patients and normal patients. Unfortunately, some of them were not clustered correctly (mixed clouds of blue and red points) in the same area.

correlation

This mixture could be, probably to the fact that many features correlate, so our next course of action was to check for any correlation between features to determine which feature are correlated by using pearson correlation

```
# Remove highly correlated variables
data_no_corr <- data[ , !apply(cor_matrix_rm,  2,function(x) any(x > 0.8))]
```

```
gl.cor2 = cor(data_no_corr, method = c("pearson"))
```

Preform PCA on our new data

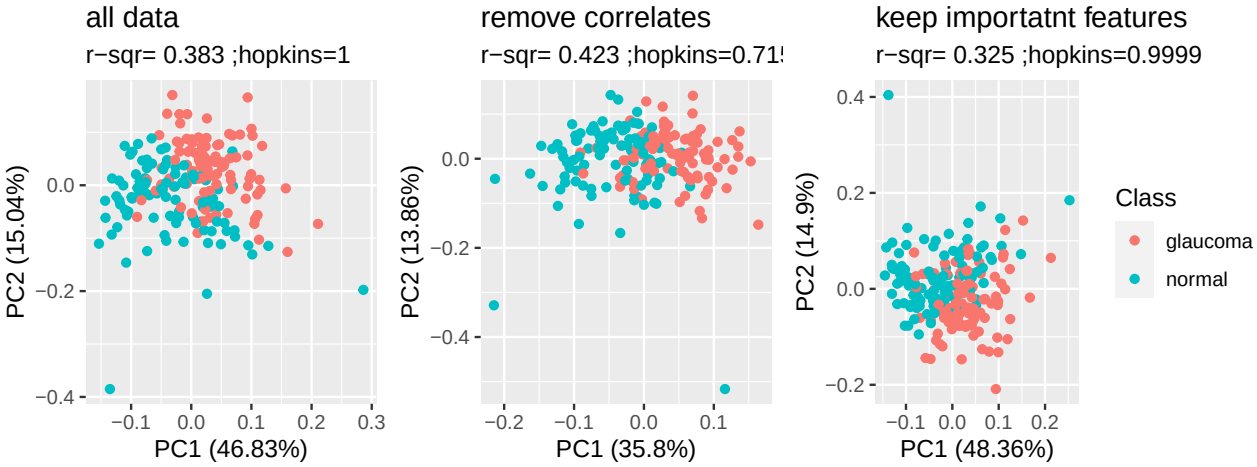
```
fit_corr_pca  <- lm(Class ~ ., cbind(Class = glaucoma$Class, corr_pca$x[, 1:2])
                    %>% as.data.frame())
```

importance

We then used the varImp to estimate the variable importance, which is printed and plotted, using the logistic regression method to train. We will start learning about training and ML algorithms starting next practice.

```
control <- trainControl(method="repeatedcv", number=10, repeats=3)
model <- train(Class~., data=glaucoma, method="multinom", preProcess="scale", trControl=control)
importance <- varImp(model, scale=FALSE) # estimate variable importance
print(importance)
plot(importance)
important_feat<-glaucoma[,rownames(importance$importance)[importance$importance$Overall>0.5]]
important_feat.pca <- prcomp(important_feat, center = TRUE,  scale. = TRUE)
summary(important_feat.pca)
important_feat[,"Class"]<-glaucoma[,"Class"]
important_feat_plot <- autoplot(important_feat.pca, data = important_feat, colour = 'Class')
fit_important <- lm(Class ~ ., cbind(Class = glaucoma$Class, important_feat.pca$x[, 1:2]) %>% as.data.fr
```

We compared the results of all the PCA's we performed:



It looks like there is no big difference this way or that. The best result we got from the data without the correlated features. We will sometimes try to run the algorithms on the smaller dataset without the correlated, but seeing as they are all fairly similar, we'll mainly work on the whole dataset.

Split into training and test sets

We split the whole data set into 80% training set (teach the algorithm by giving it data with the expected results) and 20% test set (predicting on data where the algorithm doesn't know the result, but since it is in our dataset we do know it and can evaluate how well the algorithm succeeded in predicting).

Since we know that there are about a half of each Class in the data set, we wanted to be sure that the distribution is as we defined (about 1/2 of each Class in both training and test sets should be 'normal' and

1/2 'glaucoma'), so we split the data it 2 ways.

Split using an index After shuffling, use the first 80% for training, the last 20% for test

```
set.seed(123)
index <- sample(nrow(glaucoma),round(nrow(glaucoma)*.8),replace = FALSE)
train_set <- glaucoma[index,]
test_set<- glaucoma[-index,]
train_div<-prop.table(table(train_set$Class))
testdiv<-prop.table(table(test_set$Class))
```

```
## [1] "The glaucoma/normal distribution was: train set = 0.52/0.41, and test set = 0.48/0.59"
```

Split using the function 'initial_split' We then used the 'initial_split' function, with the built in 'training' and 'testing' functions

```
patient_split <- initial_split(glaucoma, strata = Class )
data_train <- training(patient_split) # we used the default parameters of splitting size
data_test <- testing(patient_split)
train_div<-prop.table(table(data_train$Class))
testdiv<-prop.table(table(data_test$Class))
```

```
## [1] "The glaucoma/normal distribution was: train set = 0.5/0.5, and test set = 0.5/0.5"
```

Using this method we got exactly 0.5 in both sets. We will use this division, so we saved the true "Classes" of the training and test sets. We also created a training and test set that include only the desired features.

Since in most algorithms it is important for the data to be scaled, we decided to start by normalizing it.

normalization: We decided to normalize the data, 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)))}
```

```
train_set_norm <- cbind(as.data.frame(lapply(data_train[,-63], normalize)), Class=data_train$Class)
test_set_norm <- cbind(as.data.frame(lapply(data_test[,-63], normalize)), Class=data_test$Class)
summary(train_set_norm$ag) # test our dataset
```

z-score standardization: We also tried with z score scaling instead of normalization.

```
train_set_z <- cbind(as.data.frame(lapply(data_train[,-63], scale)), Class=data_train$Class)
test_set_z <- cbind(as.data.frame(lapply(data_test[,-63], scale)), Class=data_test$Class)
```

algorithms evaluation

In order to decide which algorithm was best we calculated the accuracy, sensitivity, precision and specificity. We also wanted one general number that includes all the calculations, so we did:

- Sensitivity (true predicted glaucoma/have glaucoma) is the percentage of true positives -predicted to have glaucoma- out of all the people that actually have glaucoma. In other words- it refers to the test's ability to correctly detect ill patients who do have the condition. Since we don't want to miss a sick person that then won't be treated (type II error), it is more important that the sensitivity will be high. We decided to give it a weight of 4.
- Precision (true predicted glaucoma/predicted glaucoma), or positive predictive value (PPV) is the percentage of true positives out of all the predicted positives. We don't want to scare a person that isn't sick for nothing (type I error), but of course (in case of glaucoma) that error is preferable to type II error. We decided to give it a weight of 2.

- Accuracy (true prediction/all predictions) is the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined. It's a test to generally know how well we predicted, without segregating the positives from the negatives. We decided to give it a weight of 2.
- Specificity (true predicted healthy/ healthy) relates to the test's ability to correctly reject healthy patients without a condition. It's also important, but less so. We decided to give it a weight of 1.

```
calc_score <- function(x) {return (round( ((2*x$Accuracy + 4*x$Sensitivity +
                                        2*x$Precision + x$Specificity) / 9 ),3)) }
```

SVM (Support Vector Machines)

Support vectors are the data points that define the position and the margin of the hyper-plane. They are called "support" vectors, because these are the representative data points of the classes. If we move one of them, the position and/or the margin will change. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into that same space and predicts which category they belong to based on which side of the gap they fall. The SVM algorithm maps training examples to points in space so as to maximize the width of the gap between the two categories. The effectiveness of SVM depends a lot on the selection of kernel. We tried using 4 different kernels and compared their results.

default

```
Class_classifier_svm <-  ksvm(x=Class ~., data=data_train)
Class_predictions_svm <- predict(Class_classifier_svm, data_test)
svm_linear_norm <- confusionMatrix(Class_predictions_svm,data_test[,63], dnn = c('actual', 'predicted'))
svm_stats_default <- data.frame(
                           Accuracy=svm_linear_norm[["overall"]][["Accuracy"]],
                           Sensitivity=round(svm_linear_norm[["byClass"]] [["Sensitivity"]],3),
                           Precision=round(svm_linear_norm[["byClass"]] [["Precision"]],3),
                           Specificity=round(svm_linear_norm[["byClass"]] [["Specificity"]],3))
svm_stats_default$General <- round(calc_score(svm_stats_default),3)
```

We ran the algorithm on the datatset with the selected features

```
## [1] "the whole normalized dataset's score = 0.857 > the abridged dataset's score = 0.833"
```

Interestingly, we got a worse result using the selected featured than the whole dataset. ## different kernels

Since it is important when using SVM to work only on numeric and scaled data, we made sure our data was numeric (all columns originally numeric, and the 'Class' column we factorizes), randomized (we used the 'sample' function beforehand) and scaled. The package we used (kernlab) let us scale the data when running the algorithm itself, so we'll start by scaling with the algorithm. We built the model:

```
train_svm_linear<-train(Class ~ ., data_train,
                        method = "svmLinear",
                        trControl = trainControl(method="repeatedcv", number = 10, repeats = 10),
                        metric = "Kappa",
                        preProcess = c("center","scale"))

svm_predictions_linear<-predict(train_svm_linear, data_test)
Class_classifier_linear <-  ksvm(x=Class ~., data=data_train,kernel = "vanilladot")
```

We saw that the model used 49 support vectors, and that the training error was 0.054795. SVM can be a bit of a black box so we don't know much else. After building the model on our training set (80% of the data) we ran the model on the test set:

```
Class_predictions_linear <- predict(Class_classifier_linear, data_test[,-63])
```

We used the table() function to compare the predicted Class to the true Class in the testing:

```
table(Class_predictions_linear, data_test$Class)
svm_linear <- confusionMatrix(Class_predictions_linear,data_test[,63])
```

Now we wanted to see how well our classifier performed according to the different ways of measuring success, and compared to using only the selected features.

```
## [1] "linear: the whole dataset's score =  0.81 > the abridged dataset's score =  0.786"
```

Again the whole dataset yielded better results. A standard Gaussian Radial basis function (RBF) is the second kernel we tried.

```
Class_classifier_rbf <-  ksvm(x=Class ~., data=data_train,kernel = "rbfdot")
```

```
## [1] "RBF: the whole dataset's score =  0.857 > the abridged dataset's score =  0.833"
```

```
Class_classifier_poly <-  ksvm(x=Class ~., data=data_train,kernel = "polydot")
##  Setting default kernel parameters
```

```
##  Setting default kernel parameters
```

```
## [1] "polynomial: the whole dataset's score = 0.81 > the abridged dataset's score = 0.786"
```

```
Class_classifier_tan <- ksvm(x=Class ~., data=data_train,kernel = "tanhdot")
```

```
## [1] "Hyperbolic: the whole dataset's score =  0.572 > the abridged dataset's score =  0.604"
```

We saw that again using only the selected features gave no advantage. The best result was using the RBF kernel, followed by the linear kernel:

```
##        Accuracy Sensitivity Precision Specificity General
## linear     0.82        0.76     0.864        0.88   0.810
## RBF        0.86        0.84     0.875        0.88   0.857
```

We got the same results when running the RBF as the default, and that is the best result. The linear, polynomial and hyperbolic kernels also yielded the same results, with a slight decrease in accuracy (from 86% to 82%), sensitivity (from 84% to 76%) and precision (from 87.5% to 86.4%). In both cases the specificity is 88% which is good. The polynomial kernel yielded the exact same results as the linear kernel, so we didn't continue working with it.

Other kernels might prove even better, or the cost of constraints parameter C could be varied to modify the width of the decision boundary.

pre-scaled data

normalized data

We tried using the data that we normalized and compared the 2. While using the 'ksvm' function we chose "scaled=FALSE"

```
Class_classifier_linear_norm <-  ksvm(x=Class ~., data=train_set_norm,
                                      kernel = "vanilladot", scaled=FALSE)
```

As before we ran the same kernels (except for the polynomial which yielded the same result as the linear) on both the whole dataset and the selected part, but show here the process of the linear kernel only (the rest can be seen in our code).

```
## [1] "linear: the whole normalized dataset's score =  0.8 > the abridged dataset's score =  0.8"
```

```
## [1] "rbf: the whole normalized dataset's score =  0.844 > the abridged dataset's score =  0.646"
```

Table 1: best SVM results

	Accuracy	Sensitivity	Precision	Specificity	General
RBF	**0.86**	**0.84**	**0.875**	**0.88**	**0.857**
RBF norm	0.86	0.72	1.000	1.00	**0.844**
RBF z-scaled	0.86	0.84	0.875	0.88	**0.857**

```
## [1] "tan: the whole normalized dataset's score =  0.213 < the abridged dataset's score =  0.61"
```

We saw that again using only the selected features gave no advantage. The 'General' scores were:

The best result was using the RBF kernel, followed by the linear kernel: Using the linear, RBF and Polynomial kernels we managed to got to specificity and precision of 1: all those predicted healthy were actually healthy. But it came at the price of sensitivity: the linear and polynomial yielded a sensitivity of 64%, and the RBF of 68%.

SVM results using our normalization function:

```
##              Accuracy Sensitivity Precision Specificity General
## linear           0.82        0.64     1.000        1.00   0.800
## rbf              0.86        0.72     1.000        1.00   0.844
## tan, no corr     0.58        0.68     0.567        0.48   0.610
```

Interestingly, the hyperbolic kernel on the whole dataset yielded the worst results by far, with sensitivity and precision of 0! the same kernel on the smaller dataset yielded a bit higher results, but still very bad ones (our general score jumped from 24% to 60.2%, but the other kernels are at 82% or 84%)

z-score scaled

We tried scaling with z-score as well. The rest of the code is exactly as before so we did not print it in the report.

```
Class_classifier_linear_z <-  ksvm(x=Class ~., data=train_set_z,kernel = "vanilladot", scaled=FALSE)
```

SVM results on scaled data:

```
##              Accuracy Sensitivity Precision Specificity General
## linear           0.82        0.80     0.833        0.84   0.816
## rbf              0.86        0.84     0.875        0.88   0.857
## tan, no corr     0.62        0.56     0.636        0.68   0.604
```

Again, we got the best results while using the RBF kernel, though this time it was better in all aspects.

SVM conclusions

We saw that out of all the kernels, the standard Gaussian Radial basis function (RBF) gave the best result. We also saw that using the smaller dataset that included only the selected features did not yield any better results. One last thing to note is that normalization yielded the same accuracy (86%), worse sensitivity (72% as opposed to 84%), but managed to get to 100% in both precision and specificity.

Using the normalized data, all 25 of the healthy were predicted as healthy, whereas using the default scaling 3 were told they were sick when they were actually healthy. On the other hand, out of 25 people with glaucoma, when we used the default scaling with the RBF kernel it predicted correctly 21 with glaucoma (so 4 sick people were told they are healthy), as opposed to only 18 that the normalized data yielded, (so 7 sick people think they are healthy..). The 'General' score the way we decided to calculate it gave the best result to the RBF kernel on the scaled/default data (85.7% as opposed to 84.4%), even though there was a 100% success rate in both precision and specificity, but if we calculate it differently (for instance less weight on sensitivity)

we might decide that scaling might do a better job with normalizing. In this case, we decided it's preferable that 3 people think they are sick and later on learn they are healthy if 3 less people are diagnosed correctly when they are sick, so we will choose the default as the best SVM result.

confusion matrix of the best SVM model:

	glaucoma	normal
glaucoma	21	3
normal	4	22

We saved the SVM results of our normalized data with the RBF kernel to compare with the rest of the algorithms.

K-Nearest Neighbors (KNN) algorithm

The KNN algorithm is a non-parametric supervised learning method, used for classification and regression (we used it for classification). It is based on the assumption that similar objects exist in close proximity (are near to each other). K is a positive integer (a user-defined constant) that represents the number of training samples. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class which is most frequent among the k training samples nearest to that query point.

normalized data

choose K's manually

We started by choosing different K's manually. We chose and tried k=3,9,15,21, but we show here the whole process just for k=3

```
pred_knn_3 <- knn(train = train_set_norm[,-63], test = test_set_norm[,-63], cl = data_train[,63], k=3)
knn_table_3<-CrossTable(x = data_test[,63], y = pred_knn_3, prop.chisq=FALSE)[["t"]]
knn_seperate_cv_stats <- data.frame(k=3, Accuracy=accuracy(knn_table_3)$.estimate,
                            Sensitivity=round(sensitivity(knn_table_3),3),
                            Precision=round(precision(knn_table_3),3),
                            Specificity=round(specificity(knn_table_3),3))
```

The best sensitivity result (86.4%) was for k=9, but the accuracy was 82%, the precision was 76% and specificity of 78.6%. Using our calculation of the best score, the best results we got was with k=3, with accuracy, sensitivity, precision and specificity all equal to 84%. We then ran the again algorithm with k=3, but time on the selected features only.

```
## [1] "k=3: the whole dataset's score is 0.84 > the abridged dataset's score > 0.806"
```

We got much better results with the whole dataset.

choose K's with Caret's "train"

Instead of choosing the K's manually we can use tuneGrid and find the best k in the range of 1-30

cross validation: We trained the algorithm on the normalized training set with the 'train' function of the 'caret' library with different options of K (in the range of 1-30). We started with cross validation.

```
knn_train_cv<-train(Class ~ .,train_set_norm, method = "knn",
                        trControl = trainControl(method = "cv", number = 10),
                        tuneGrid = data.frame(k=seq(1,30,2)),
                        metric = "Kappa")
```

We used confusionMatrix to check how well we did.

```
knn_predictions_cv<-predict(knn_train_cv, test_set_norm)
knn_cv_mat <- confusionMatrix(knn_predictions_cv,data_test[,63])
```

We again did the same thing just on the dataset that didn't include high correlations.

```
## [1] "CV: the whole dataset's score = 0.833 > the abridged dataset's score = 0.842"
```

The sensitivity was higher on the smaller dataset (80%->84%) with one less sick diagnosed as healthy, and the accuracy stayed the same, but the precision (87%->75%) and specificity (88%->72%) both decreased drastically (4 more healthy were diagnosed as sick), so the overall grade was higher for the dataset that didn't include high correlations.

Compared to the normalized manually chosen data we have an increase in both precision (84%->87%) and the specificity (84%->88%), but a decrease in sensitivity (84%->80%). Because we put so much weight on the sensitivity, even though the precision and specificity were higher (the accuracy stayed 84%), the overall general score was lower.

repeated cross validation: We wanted to see if using repeated cross validation will yield better results.

```
knn_train_repeatedcv<-train(Class ~.,train_set_norm,
                        method = "knn",
                        tuneGrid = data.frame(k=seq(1,30,2)),
                        trControl = trainControl(method = "repeatedcv", number = 10, repeats = 10),
                        metric = "Kappa")
```

```
## [1] "repeated CV: the whole dataset's score = 0.754 < and the abridged dataset's score = 0.842"
```

This time there was a preference toward the whole dataset, with all evaluations increasing with the smaller dataset.

We wanted to see if we get better results if we use scaling on our data instead of normalizing. Between CV and repeated CV, on the smaller dataset we got better results using repeated CV, but they were both not as good as the one from the whole dataset, which there was no difference between the 2, so we will run the algorithm on the scaled dataset using repeated cross validation (and choosing manually, which so far yielded the best results).

z-score standardization

manually

We then tried with z score scaling instead of normalization. Again we started by choosing manually, and tried k=3,9,15,21, though we show here just for k=3

```
pred_knn_3_z <- knn(train = train_set_z[,-63], test = test_set_z[,-63], cl = data_train[,63], k=3)
knn_table_3_z<-CrossTable(x = data_test[,63], y = pred_knn_3_z, prop.chisq=FALSE)[["t"]]
```

We got the best 'General' result from k=3, with precision of 88% and specificity of 87%, but with sensitivity of 81.5%. The best sensitivity (84%) we got with k=21, but with precision and specificity of 84%. We wanted again to check if we can get better results with the smaller dataset. We decided to run it only with k=3, since that was the best result according to our calculations.

```
best_knn_stats[paste(
  "manually", knn_seperate_cv_stats_z_new$k,", scaled, selected"),] <-
  knn_seperate_cv_stats_z_new
```

We saw that the results were much better using the z-score scaling. The best result was from k=3, with accuracy, sensitivity, precision and and specificity all are 92% (out of 25 sick 23 were predicted sick, out of 25

Table 2: best KNN results

	Accuracy	Sensitivity	Precision	Specificity	General
manually chosen K's	0.84	0.84	0.840	0.84	**0.840**
CV, selected	0.84	0.80	0.870	0.88	**0.833**
repeated CV	0.74	0.80	0.714	0.68	**0.754**
manually k=3 , scaled, selected	**0.92**	**0.92**	**0.920**	**0.92**	**0.920**
CV, scaled, selected	0.84	0.84	0.840	0.84	**0.840**

healthy 23 were predicted healthy) We then proceeded to check the cress validation using Caret again but on scaled data

repeated cross validation on scaled data

```
knn_train_repeatedcv_z<-train(train_set_z[,-63],data_train[,63],
                        method = "knn",
                        tuneGrid = data.frame(k=seq(1,30,2)),
                        trControl = trainControl(method = "repeatedcv", number = 10, repeats = 10),
                        metric = "Kappa")
```

```
## [1] "CV: the whole z-scaled dataset's score using is 0.833 < the abridged dataset's score = 0.84"
```

Again we got better results out of the smaller dataset (accuracy from 84%0>86%, sensitivity 80%->84% and precision 87%->87.5%)

knn conclusions

We did not see any difference whether we used the crossed validation or the repeated cross validation. We got worse results when we used the smaller dataset on the normalized data, but much better on the scaled data. Interestingly, the best results were with the manually chosen K's (when K=3). using CV and repeated CV the best result was with K=17. We had manually chosen K=15 and K=21, and both were not as good as K=3. Using KNN algorithm with K=3 on scaled data with only the selected featured gave us the best result yet.

confusion matrix of the best KNN model:

	glaucoma	normal
glaucoma	23	2
normal	2	23

Decision Tree (DT) algorithm

Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely decision nodes and leaves (cf https://www.xoriant.com/).

```
DT_model <- C5.0(data_train[-63], data_train$Class)
```

The preceding text shows some simple facts about the tree, including the function call that generated it, the number of features (labeled predictors), and examples (labeled samples) used to grow the tree. Also listed is the tree size of 62, which indicates that the tree is 62 decisions deep.

Next we looked at the summary of the model.

```
summary(DT_model)
```

The numbers in parentheses indicate the number of examples meeting the criteria for that decision, and the number incorrectly classified by the decision. For instance, on the first line, 56/4 indicates that of the 56 examples reaching the decision, 4 were incorrectly classified as not likely to default. In other words, 4 applicants actually defaulted, in spite of the model's prediction.

```
DT_pred <- predict(DT_model, data_test)
DT_pred_tbl<- confusionMatrix(DT_pred,data_test[,63])
```

```
##            Accuracy Sensitivity Precision Specificity
## DT default     0.88        0.84     0.913        0.92
```

The performance here is somewhat worse than its performance on the training data, but not unexpected, given that a model's performance is often worse on unseen data. Also note that there are relatively many mistakes where the model predicted not a default, when in practice the loaner did default. Unfortunately, this type of error is a potentially costly mistake, as sick patients could be blind with time. We will try to improve the result.

Adaptive Boosting

To improve our model we will use a C5.0 feature called adaptive boosting. This is a process in which many decision trees are built and the trees vote on the best class for each example.

The C5.0() function makes it easy to add boosting to our C5.0 decision tree. We simply need to add an additional trials parameter indicating the number of separate decision trees to use in the boosted team. The trials parameter sets an upper limit; the algorithm will stop adding trees if it recognizes that additional trials do not seem to be improving the accuracy. We started 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(data_train[-63], data_train$Class , trials = 10)
summary(DT_boost10)
```

```
##                          Accuracy Sensitivity Precision Specificity
## boosting with 10 trials     0.84         0.8      0.87        0.88
```

The classifier got an error rate of 0% percent. This is quite an improvement over the previous training error rate before adding boosting! However, the model is still not doing well at predicting class of patients, which may be a result of our relatively small training dataset, or it may just be a very difficult problem to solve.

fine-tune with cost-matrix

Next, we proceeded to fine-tune our algorithm, using a cost matrix. The C5.0 algorithm allows us to assign a penalty to different types of errors, in order to discourage a tree from making more costly mistakes. The penalties are designated in a cost matrix, which specifies how much costlier each error is, relative to any other prediction.

We created a default 2x2 matrix, and filled with our cost values. If we consider that the most important is to predict a sick patient as sick, much more that a normal patient as normal, our penalty values could then be defined as:

```
##           actual
## predicted  normal glaucoma
##   normal        0       20
##   glaucoma      6        0
```

We trained again to see if the cost matrix made any difference.

Table 3: best DT stat

	Accuracy	Sensitivity	Precision	Specificity	General
DT default	**0.88**	**0.84**	**0.913**	**0.92**	**0.874**
boosting with 10 trials	0.84	0.80	0.870	0.88	**0.833**
fine tuning with cost matrix	0.72	0.84	0.677	0.60	**0.750**

```
dt_cost <- C5.0(data_train[-63], data_train$Class , costs = error_cost)
```

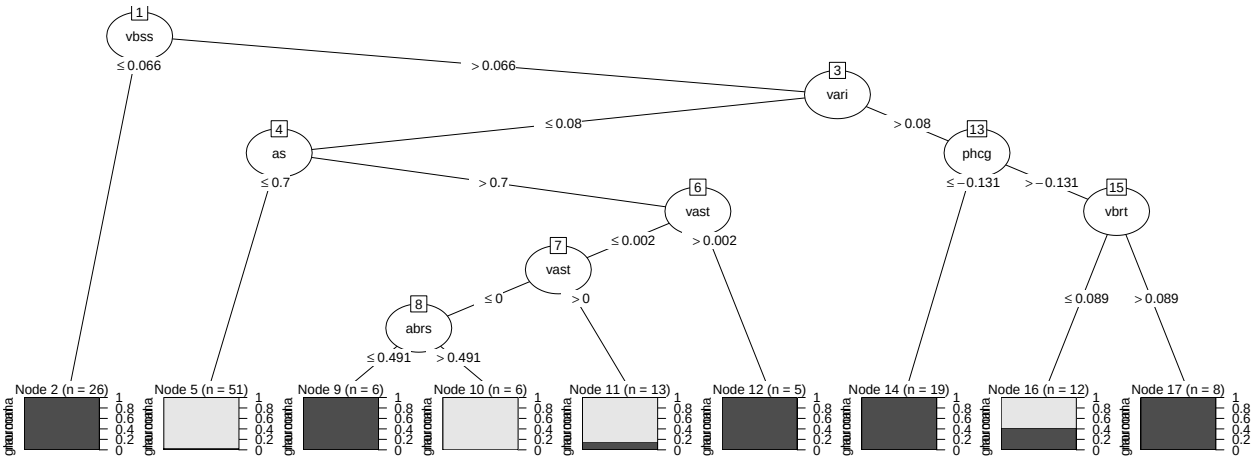


Figure 1: Desition Tree

This last version makes more mistakes overall, but the types of mistakes are very different. Where the previous models incorrectly classified a small number of class correctly, our weighted model does much better in this regard. This trade resulted in a reduction of false negatives at the sick patient false positives may be acceptable if our cost estimates were accurate.

DT conclusions

Our best model was the first, default model, with sensitivity of 84% (like with the cost-matrix) and precision of 91.3% and specificity of 92%.

confusion matrix of the best DT model:

	glaucoma	normal
glaucoma	21	2
normal	4	23

Random forest (RF) algorithm

Random forest is a Supervised Machine Learning Algorithm that is used in Classification and Regression problems. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression. The Random Forest Algorithm can handle the data set containing continuous variables as in the case of regression and categorical variables as in the case of classification.(cf: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/)

Random Forest Modeling

The random Forest algorithm uses many decision trees. For classification, which is what we are focusing on, the final class result is based on the class that was picked from most trees.

First we define our model with the parsnip package. Random forests have very little mandatory parameters. Here we only define the number of trees:

```
rf_mod <-  rand_forest(trees = 1000) %>%  set_engine("ranger") %>% set_mode("classification")
```

We want reproducible results, so we set.seed. Taking a look at the model:

```
rf_fit <- rf_mod %>% fit(Class ~ ., data = data_train) # train model
```

We can see our model had 1000 trees like we specified, and several other metrics, such as the prediction error, 13% here.

Estimating performance

Next we want to see if we can improve our model. We do this by changing some things about it. The changes can either be random (guessing) or more precise, depending on performance metrics. In our example, we will use the area under the Receiver Operating Characteristic (ROC) curve (which demonstrates the trade-off between the sensitivity and and specificity), and overall classification accuracy.

Using the yardstick package, let's calculate ROC and Accuracy. Notice that we are still only working with the data_train partition of our data:

```
rf_training_pred <- predict(rf_fit, data_train) %>%
  bind_cols(predict(rf_fit, data_train, type = "prob")) %>%
  bind_cols(data_train %>%  select(Class))
```

```
## [1] "training: the roc_auc estimate =  0.999 , the accuracy estimate =  0.979"
```

As we can see, these are very good results. Almost too good, but it is on the set we trained on. There are several reasons why training set statistics like the ones shown in this section can be unrealistically optimistic:

- Overfitting - Models like random forests, neural networks, and other black-box methods can essentially memorize the training set. Re-predicting that same set should always result in nearly perfect results.
- The training set does not have the capacity to be a good arbiter of performance. It is not an independent piece of information; predicting the training set can only reflect what the model already knows.

so we checked how the model performs on the test data:

```
rf_testing_pred <-
  predict(rf_fit, data_test) %>%
  bind_cols(predict(rf_fit, data_test, type = "prob")) %>%
  bind_cols(data_test %>% select(Class))
```

```
## [1] "testing: the roc_auc estimate =  0.934 , the accuracy estimate =  0.82"
```

These validation results are lower than the ones we got on the training data, as expected, but are still pretty good.

Resampling

Resampling methods, such as cross-validation and the bootstrap, are empirical simulation systems. They create a series of data sets similar to the training/testing split discussed previously; a subset of the data is used for creating the model and a different subset is used to measure performance. Resampling is always used with the training set.

Here we'll use 10-fold cross-validation. This means that we'll create 10 "mini" datasets, or folds. We call the majority part of the folds (9 out of 10 in this case) the "analysis set" and the minority the "assessment set". We then train a model using the analysis set, and test it on the assessment set, effectively repeating the modeling process 10 times. This is how its done:

```
folds <- vfold_cv(data_train, v = 10) # create the folds
rf_wf <-   workflow() %>%  add_model(rf_mod) %>%  add_formula(Class ~ .)
rf_fit_rs <-   rf_wf %>%   fit_resamples(folds)# add folds to workflow and train model
```

```
## [1] "the accuracy estimate = 0.829 , the  roc_auc estimate =  0.903"
```

We see these results are lower and look more realistic.

Tuning hyperparameters

Another way to improve the performance of our models is by changing the parameters we provide them with. This is also called Tuning. Random Forests, as mentioned above, are not very sensitive to such parameters, but decision trees are. Lets see:

```
# defining the parameters for the decision tree
tune_spec <-   decision_tree(cost_complexity = tune(), # to control the size of the tree
    tree_depth = tune()) %>% set_engine("rpart") %>%   set_mode("classification")
```

Note that in the above code, tune() is still just a placeholder. We will fill in values later on.

Next, we will create several smaller datasets, to try different parameters. The grid_regular command below will create 5 such datasets for each parameter combination, meaning 25 in total.

```
tree_grid <- grid_regular(cost_complexity(), tree_depth(), levels = 5)
tree_grid %>%   count(tree_depth)
glaucoma_folds <- vfold_cv(data_train) # create the actual cross-validation folds
```

Model tuning with a grid

```
tree_wf <- workflow() %>%  add_model(tune_spec) %>%  add_formula(Class ~ .)
tree_res <- tree_wf %>%  tune_grid(resamples = glaucoma_folds, grid = tree_grid)
tree_res %>% collect_metrics()
```

Its easier to see how the models did with a graph:

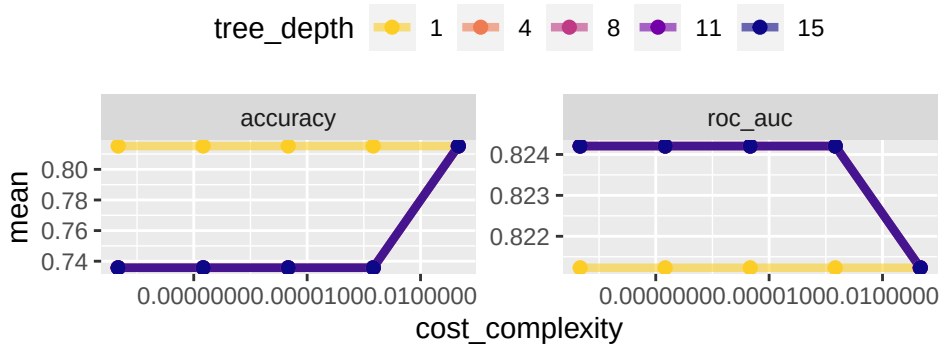


Figure 2: Model tuning with a grid evaluation

We can see that our tree with a depth of 1, is the worst model according to both metrics and across all candidate values of cost_complexity. Our deepest tree, with a depth of 15, did better. The show_best() function shows us the top 5 candidate models by default:

Table 4: best RF stat results

	Accuracy	Sensitivity	Precision	Specificity	General
RF default	0.82	0.80	0.833	0.84	**0.816**
10-fold CV	0.84	0.80	0.870	0.88	**0.833**
final_fit	**0.86**	**0.84**	**0.875**	**0.88**	**0.857**

And finally, let's finalize the workflow with the best tree.

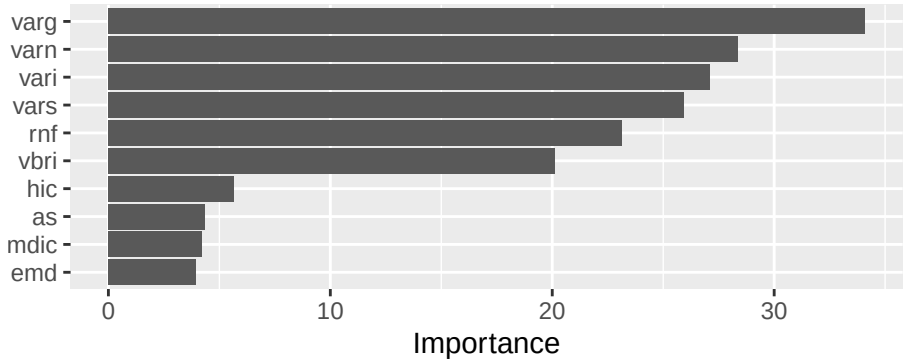
```
best_tree <- tree_res %>%  select_best("roc_auc") # Selecting the best tree
final_wf <-   tree_wf %>%   finalize_workflow(best_tree) # Finalize workflow
print(final_wf)
```

```
##   cost_complexity tree_depth               .config
## 1           1e-10          4 Preprocessor1_Model06
```

So we have our best tree parameters. Lets take a better look:

```
# Exploring results
final_tree <-   final_wf %>%  fit(data = data_train)
print(final_tree)

# vip package: another way to look at how the model uses the different features
final_tree %>%   extract_fit_parsnip() %>%   vip()
```



Random forest conclusions and Finalizing the model

Finally, let's return to our test data and estimate the model performance we expect to see with new data. We can use the function last_fit() with our finalized model; this function fits the finalized model on the full training data set and evaluates the finalized model on the testing data.

```
final_fit <- final_wf %>%  last_fit(patient_split) # Set workflow
final_fit %>%  collect_metrics()# train
final_fit_tbl_stats<-collect_metrics(final_fit)
```

```
## [1] "the accuracy estimate = 0.86 , the  roc_auc estimate =  0.866"
```

confusion matrix of the best RF model:

	glaucoma	normal
glaucoma	21	3
normal	4	22

We can observe in the ROC curve that in our true positive prediction we have very few false positive predictions: more than 75% of chances that the model distinguish sick patients to healthy patients.

Random forest classify patients with more sensitivity than decision tree. This result is not surprising by the fact that random forest build many decision trees and select the best.

conclusions

To summarize, our dataset is composed of 196 patients:98 sicks and 98 healthy. There are 63 features corresponding to the blood vessels eye pressure which help to determine if a patient has glaucoma. Each feature has its degree of importance, some of them seem to be more important than others. Keeping only important features (based on threshold determine by varImp), we did not observe any improvement in the patient classification. Moreover, we observed that some features highly correlate. Unfortunately, after deleting highly correlated features, we did not observe a major improvement in the patient classification. Thereby, we decided to keep all 63 features. We did try in the SVM and KNN algorithms also to use the selected features. We saw no improvement in the SVM model, though our best result was with the KNN algorithm using the selected features only. Before we tested the different algorithms, we defined a weight for each variable allowing to evaluate the predictions: 4 for sensitivity, 2 for precision, 2 for accuracy and 1 for specificity, in order to classify results of the algorithm used.

	Accuracy	*Sensitivity*	*Precision*	*Specificity*	*General*	*notes*	*AUC*
SVM	0.86	0.84	0.875	0.88	0.857	RBF	0.86
KNN	**0.92**	**0.92**	**0.920**	**0.92**	**0.920**	**manually k=3 , scaled, selected**	**0.92**
DT	0.88	0.84	0.913	0.92	0.874	default	0.88
RF	0.86	0.84	0.875	0.88	0.857	final fit	0.86

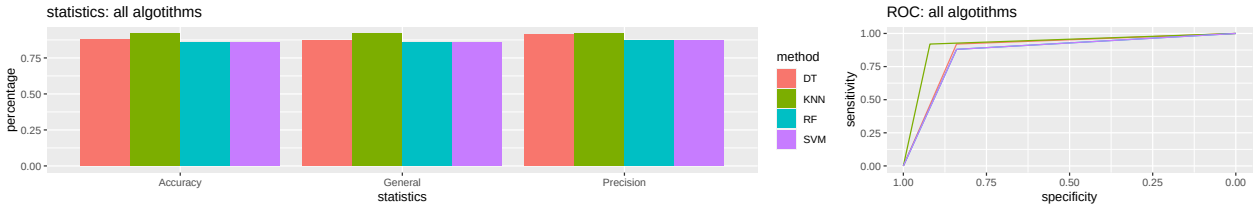


Figure 3: all statistics

The last figure shows all statistics according to algorithms (showing the best result of each algorithm) on the left, and a ROC plot (sensitivity by specificity) on the right. We observe that KNN algorithm gives the best sensitivity score (0.92), so the highest rate of true positive cases, which is the most important variable.About the precision, (true predicted glaucoma/predicted glaucoma), type I error, KNN algorithm gives the better score (0.92) followed by DT algorithm (0.913). About the accuracy (true prediction/ all prediction), proportion of correct prediction, KNN gives the best score again (0.92) followed by DT algorithm (0.88). Finally, about the specificity, (true predicted healthy/healthy), KNN and DT algorithm give the same score (0.92). Combining all together, KNN algorithm gives the best predictions followed by decision tree algorithm. SVM and random forest give similar score of predictions, lower than KNN and decision tree.

The upper table is the statistic results of all algorithms (in numbers). We can see that the SVM ans the RF yielded the same result, which was the lowest. Nest we have the DT, which has the same sensitivity, a bit higher accuracy, and much higher precision and specificity. The specificity (92%) was highest in both the DT and the KNN, and in all aspects the highest result we got was when using the KNN algorithm on scaled, selected (without highly correlated data) data.