Data Mining Project

Christian Bakke Venneroed

October 2019

1 Data preparation

Since real-world data often are incomplete or lacks certain values etc, we must process the data and convert it into an understandable format, which is what I have done. As it is easier to work with dataframes in R, the .csv file is read in and converted to a dataframe in the function fileReader(). After that, I change the column-names of the newly generated dataframe in the function change_col_names(). Missing values in the column AG_Ratio is handled in the function fix_AG_RatioCol(), where I first compute the median of the remaining elements then replace all empty values in the column with that median. After replacing all two's with zero's and changing the type of the column Class to factor, I save the processed dataframe in the function save_data_frame(). There, I use the function saveRDS(), which saves the file as a dataframe object.

2 Clustering

The now processed file is loaded in again using the function readRDS(). Before clustering, we remove some of the columns and rescale. If the data has attributes which varies very much in size or are much larger than other attributes, then these can have a undesirable effect on the clustering result[1]. Hence, it is important to normalize the data. This is done with Min-Max normalization in the function normalize(), transforming every value in the dataframe to the interval (0,1). Now, our data is ready to be clustered.

2.1 K-means

K-means is a clustering algorithm that forms k-clusters by assigning each point to its nearest centroid using a distance function, for instance euclidean distance. For each iteration, the k centroids are recalculated based on the mean of the points belonging to a cluster. Furthermore, each centroid is recalculated until the centroids dont change or a stopping criteria has been reached.

Task 2.3) Here, we use the R-function kmeans(df,k), where df is a dataframe and k is the number of clusters. Clustering with k=2 and default parameters, we plot the cluster as a 2D plot with x = Alkphos and y = TP. In order to do so, we implement the library ggplot and use the function ggplot(data,aes(x,y)), where data is the kmeans data and x and y is the attributes we are plotting against. In order to not get downscaled axes, we use the before-scaled dataframes in our plotting to get ordinary axes.

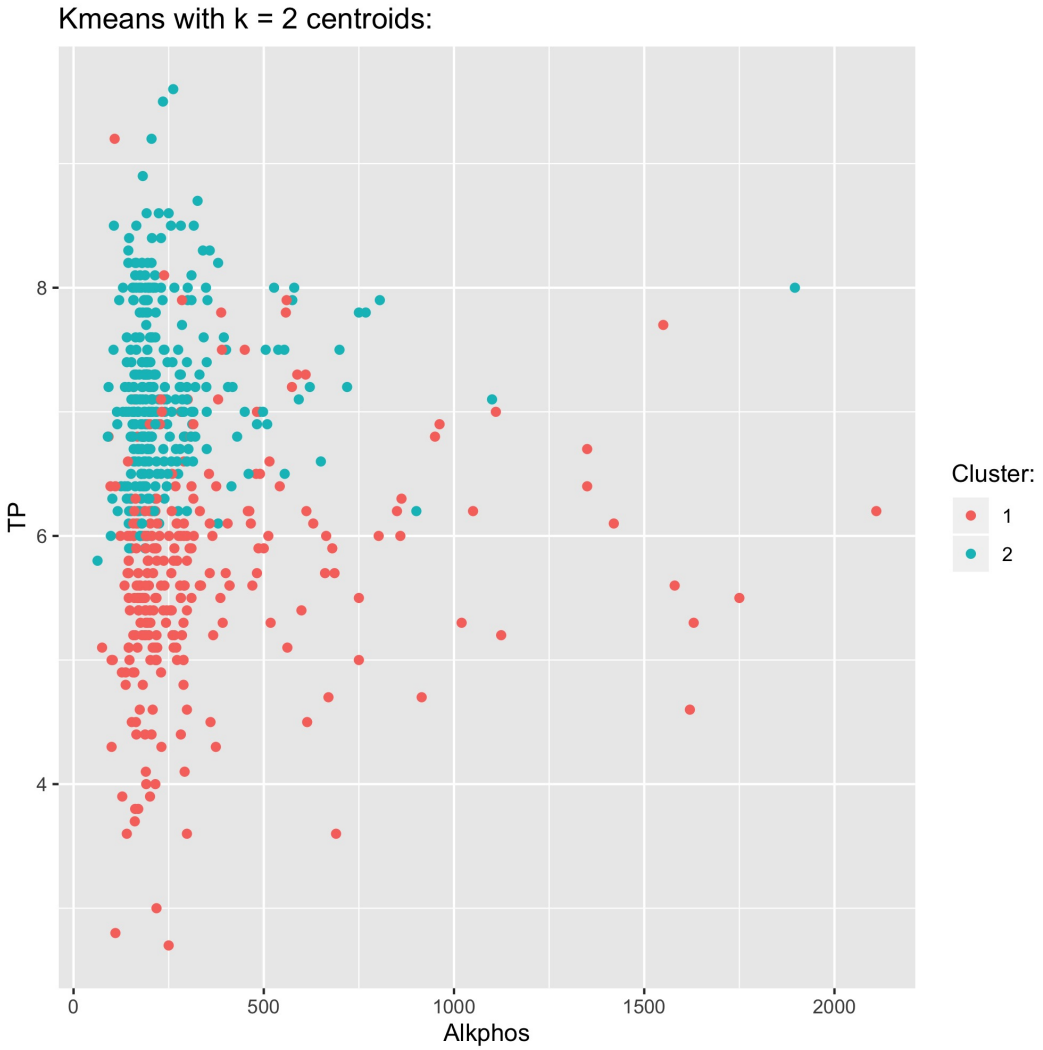


Figure 1: SSE = 43.4

As one can see from 12, the clusterings appear to be of roughly the same size. We observe that Alkphos has little effect on separating the clusters. The majority of the datapoints belonging to cluster 1 has a TP value smaller than 6, whereas almost all of the datapoints belonging to cluster 2 has a TP value larger than 6. Hence, TP may be the attribute that differentiates the clusterings, whereas other attributes may lie closer to each other.

Task 2.4) In this task, we do the same but where we color each point according to the class column.

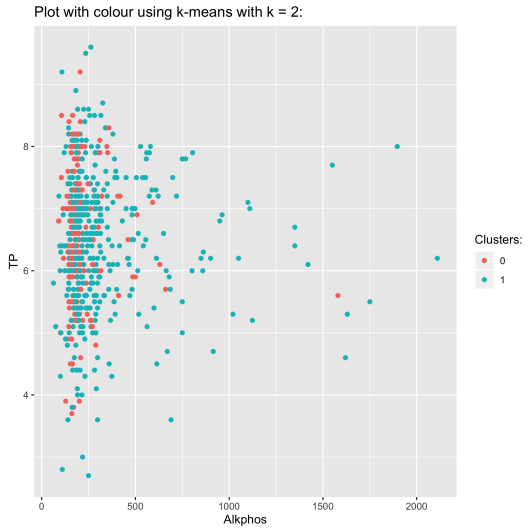


Figure 2: SSE = 43.4

From figure 2, we see that clustering using K-means with k = 2 provides no good insights; there's no clear relation between Class, Alkphos or TP. The distribution of the actual Class, Patient vs Non-patient, is evenly distributed in the plot.

Task 2.5 In order to compare 12 and 2, I've created a new plot which combines both the class-label and the clusters.

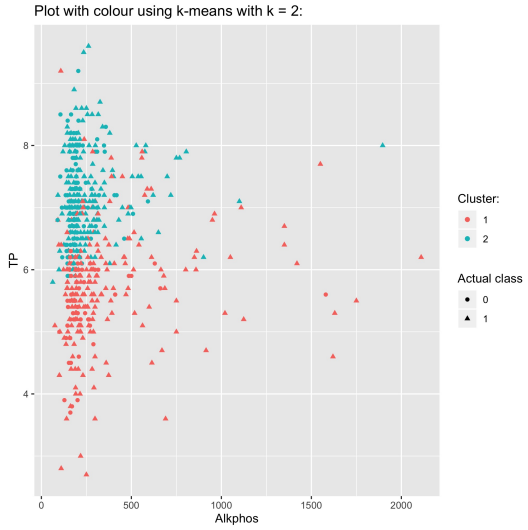
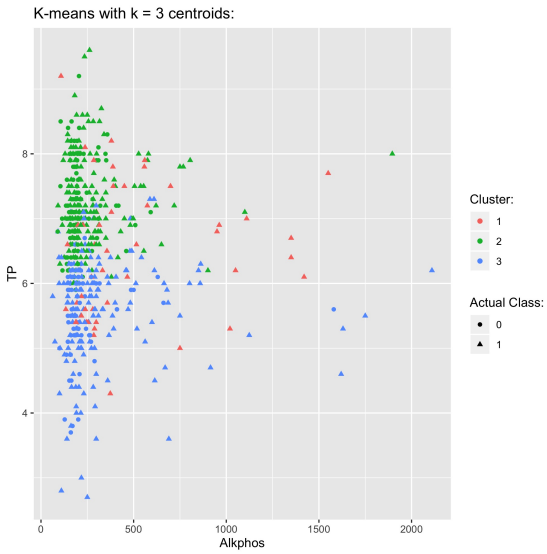


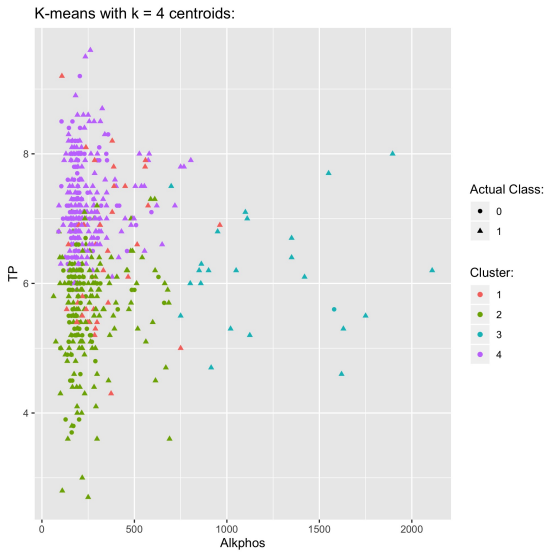
Figure 3: SSE = 43.4

Here, we see that there is no clear relation between what cluster each point has been assigned to and the actual class. Hence, by using the external index when observing the cluster validity, we see that K-means with k = 2 may not be the most suitable algorithm for this case. Also, using the internal index, for instance the sum of squared errors, also gives little insight as we have little to compare the value with.

Task 2.6 Here, we again use the K-means algorithm, this time with higher values for k. This gives the following plots and class distributions in different clusters for each k:



SSE = 31.97



SSE = 28.07

		Class	
		0	1
Cluster	1	0	50
	2	102	194
	3	65	172

		Class	
		0	1
Cluster	1	0	48
	2	63	166
	3	38	99
	4	66	103

Table 1: Class distributions in clusters using kmeans with k = 3 and 4 respectively

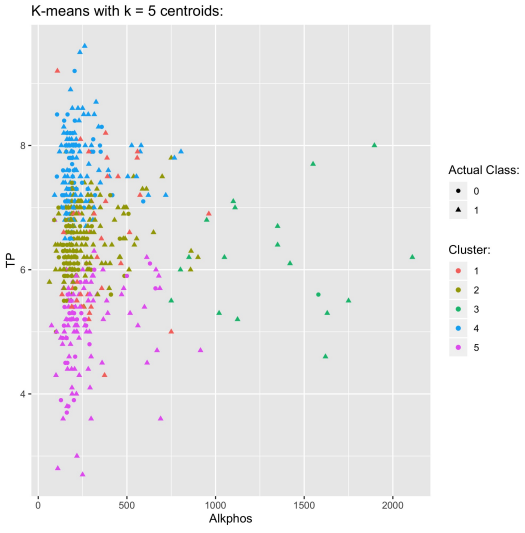
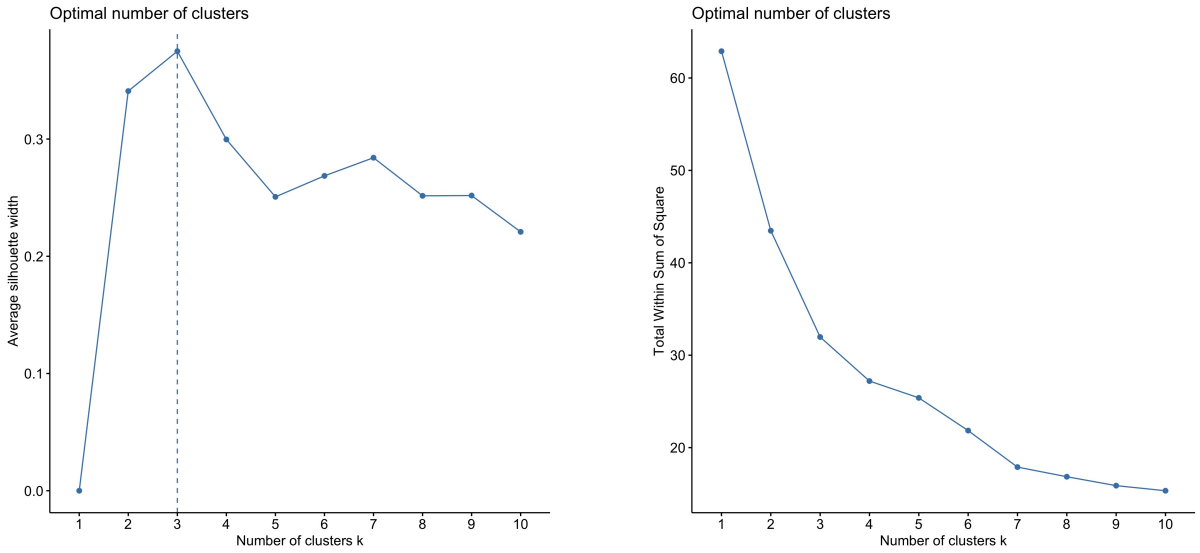


Figure 5: SSE = 23.46

		Class	
		0	1
Cluster	1	63	164
	2	38	96
	3	0	44
	4	66	102
	5	0	10

Table 2: Class distribution in clusters using kmeans with k = 5

Task 2.6 One disadvantage with the K-means clustering algorithm, is that the value for k is something the user has to choose self. Hence, different values for k must be attempted in order to select the right k.
Furthermore, in order to measure the performance of the K-means clusterings, one can utilize different metrics. First and foremost, one can compare with the external index, i.e. ground truth. In this task, we are asked to compare with internal index, so I use the metrics Sum of Squared Errors and average Silhouette width. This is done in the function sse_finder(), where I am using the library factoextra and the function fviz_nbclust(df,kmeans,method) twise, where df is the normalized dataframe, kmeans is the partitioning function and method is Silhouette and SSE respectively. The resulting plots are the following:



A: Silhoette

B: SSE

Figure 6: Selecting K

As we can see from figure 6a, k = 3 seems like the optimal number of clusters. This is also something we can see from the plots in task 2.6), the SSE falls from 43.4 with k = 2 to 31.97 with k = 3. This is typical; the average distance to each centroid often falls rapidly until optimal value of k, then it flattens, which we can also see here with k = 4 and SSE = 28.07, decreasing only 30 percent of the difference between k = 2 and k = 3. Another observation of the clustering is that with k = 3, we have one cluster which only contains patients. Perhaps that might be a subdisease?

2.2 Agglomerative clustering

With agglomerative clustering, each point is initially treated as an individual cluster. Then, until one or k clusters is left, the closest pair of clusters is merged. One advantage is that the user do not have to think about the value of k.

Task 2.8 In order to apply hierarchical clustering, we sample 50 random rows from the dataframe, and use the function hclust() with default parameters. Then, we cut the tree for a given k using the function cutree(). The tree is then plotted using the function plot().

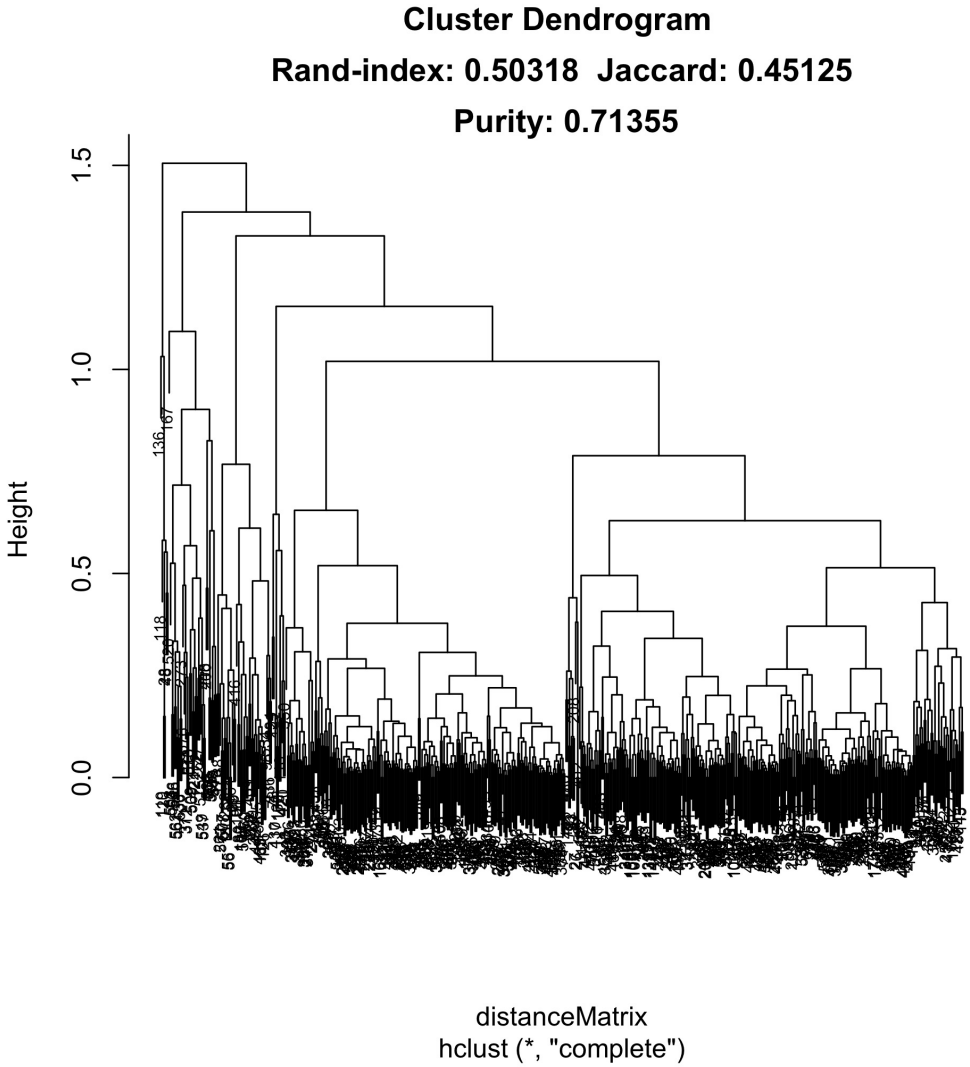
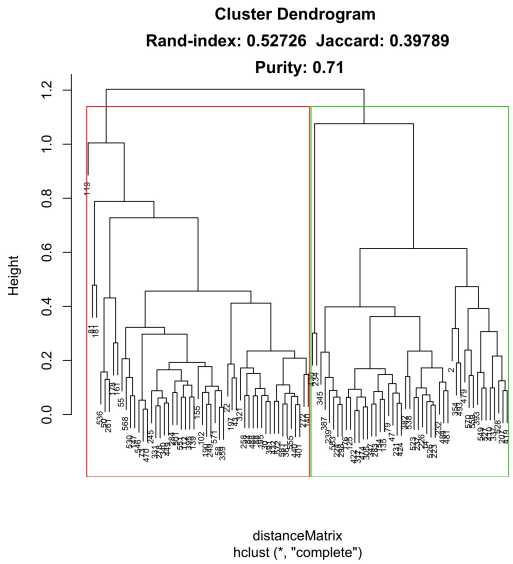
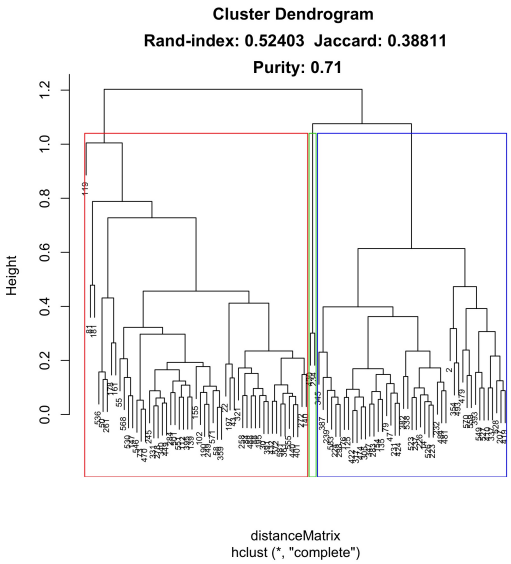


Figure 7: Hierarchical clustering on the full dataset

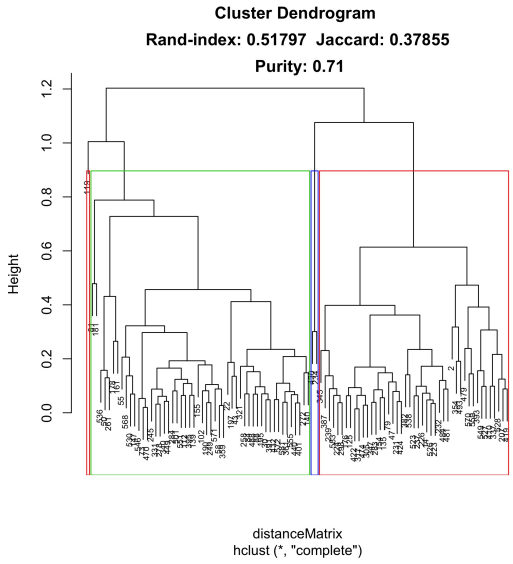
Naturally, plotting the dendrogram corresponding to hierarchical clustering on the full dataset results in a futile plot; it is very hard to draw conclusions from the dendrogram. The same would apply if we were to cut the dendrogram at for instance k = 3 or k = 5. Hence, we randomly sample 100 instances from the dataframe and instead plot them:



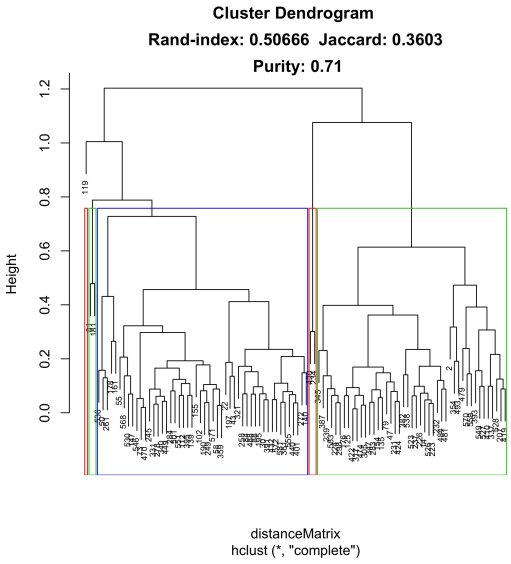
Using k = 2 and default parameters.



Using k = 3 and default parameters.



Using k = 4 and default parameters.



Using k = 5 and default parameters.

Here, I have clustered with default parameters and cut the tree for values of k = (2,3,4,5), respectively, and then plotted. We observe that the purity remains identical when k increases, whereas the purity and the Rand-index falls slightly.

Task 2.9 The observation from task 2.6) is used, where we got one cluster that only contained patients (50 of them). Then, by using the fact that k = 3 was the best value for k, I played around using different attributes on the x-axis and the y-axis, plotting and comparing against ground truth. This resulted in the following plot, where I use x = TB and y = DB, which gives this:

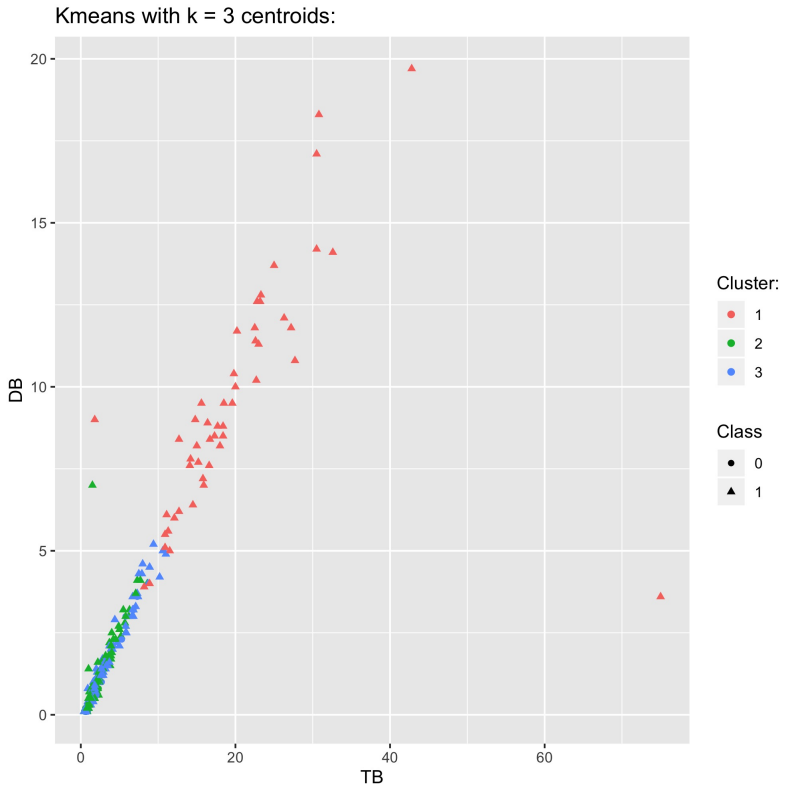


Figure 10: 2D-plot with x = TB and y = DB

As one can see from the plot, every instance with TB larger than 10 is a patient. The same applies for DB; every instance with DB larger than 5 is a patient.

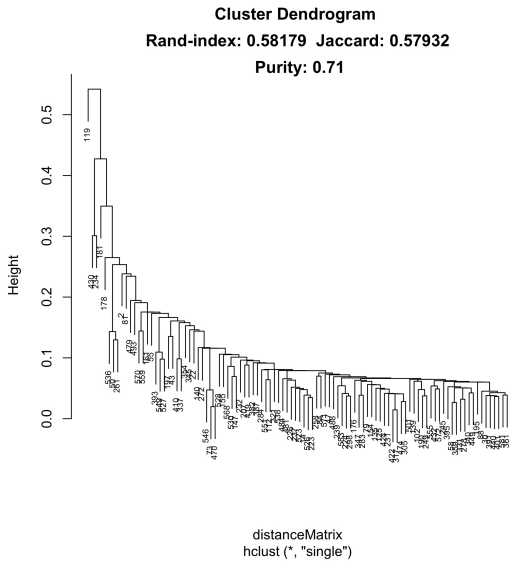
Still using k = 3, by checking all of the data samples with hierarchical clustering with MAX and comparing with ground truth, i got the following results:

Cluster:	Not-patient	Patient
1:	0	35
2:	167	375
3:	0	6

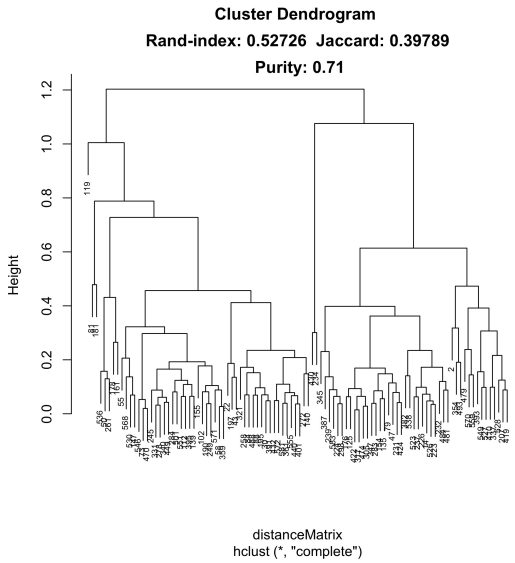
By checking with larger k, I always got one cluster that only had patients. I then extract the Row-ID's from here and the Row-ID's in Cluster 1 in figure 10, using the function compareRows(df1,df2), which takes in two dataframes, converts the Id of the rows to vectors using as.numeric(row.names(dataframe)) and then compare them using intersect(). Using this, I found that 34 out of 35 patients in the hclust-cluster, was also in Cluster 1 in figure 4a. This means that both of the clusters recognizes a pattern within the Patient-instances, meaning that there might be a subtype of the disease.

Task 2.10

Cluster validity: In order to compare the accuracy of the clustering for each agglomeration method, we use the external index, where we compare the clustering with ground truth, i.e. the actual class. To do this, I have created a function cluster_validation(), which for each row in the dataframe combines the actual class and the cluster that row belongs to. Then, iterating through the dataframe, I compute the Rand-Index, the Jaccard-coefficient and the Purity. This is done to easier being able to compare the clustering results.



Using MIN



Using MAX

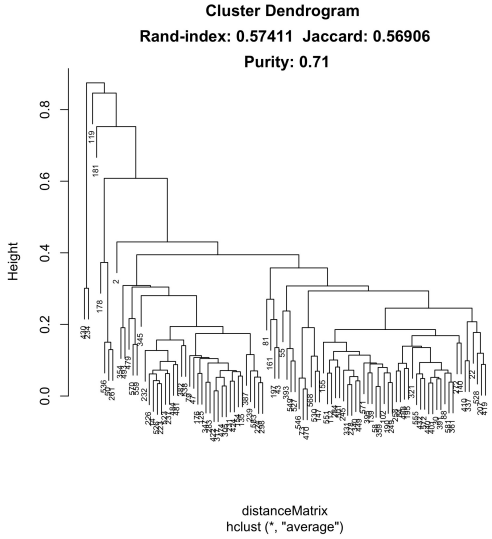


Figure 12: Using Average

We observe that the Rand-index and the Jaccard-coefficient is approximately the same using MIN and Average, but substantially lower using MAX. This might be because max tends to break large clusters.

Furthermore, we observe that the clustering using MIN appears to suffer from the chaining phenomenon, where clusters may be forced together due to single elements being close in proximity [2]. Hence, I find that Average gives best quality on this dataset.
Nevertheless, comparing with the plots in 2.8) we see with the naked eye that the default function is "complete", i.e. MAX.

3 Classification

Classification is a data mining technique, so that given a collection of records of the form (x,y), we want to find a model f so that f(x) = y, i.e. so that new records are assigned a class as accurately as possible.

Classification revolves around a twostep process; the model must first be build (model learning), before it then can be tested (model testing).

3.1 Data Partitioning

Task 3.1 Because classification is a twostep process, we must partition the data into trainingdata and testdata.

Because we were supposed to use all of the columns in this task, I loaded in the processed file using readRDS(filename), and then normalized the columns using MinMax-normalization. In order to normalize the data, the Gender column had to be converted to a numeric (female = 0, male = 1). This was done using the function within(dataframe,Gender - as.numeric(Gender)).

Now, the data was ready to be partitioned. The data was partitioned into 70 percent trainingdata and 30 percent testdata. This was done in the function partition_data(), where I first sample n numbers in the interval between one and the number of rows in the dataframe. This was done in the function partition_data(), where i sample using the built-in function sample(val,n), where val is the sampling range, i.e (1:val), and n is the number of numbers that were sampled.

3.2 Learning a Classification Tree

Task 3.2 In order to learn a classification tree, I imported the party-library. The function ctree(formula,data) was then used in order to train the model, where the parameters was set to train the model on all attributes in the data. I then plotted the result using the function plot(model,type="simple"). In order to evaluate the models performance, we must test the model on the testdata. This is done using the function predict(model,testData), which can be further combined with the actual class in order to find the confusion matrix and thus measure the tree's performance. This gave me the following results:

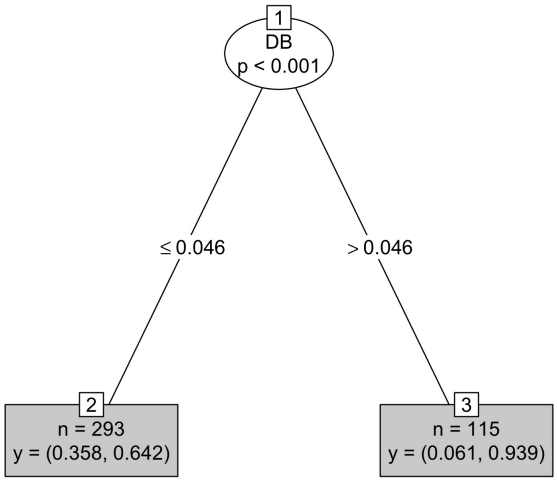


Figure 13: Ctree with default parameters (mincriterion = 0.95)

The resulting tree is quite simple, though indicating that DB is an important variable. If the null hypothesis is that the left and the right nodes of a split is the same, we want to reject that null hypothesis with high confidence, i.e. as low p-value as possible. Hence, we seek to split on variables that make the sub-nodes as different as possible [3]. By default, the ctree() algorithm rejects attributes having lower p-value than 0.05. We see that the p-value of DB is very low, and that it sends almost 1/4 of the instances to node 3.

		Prediction	
		0	1
Class	0	0	55
	1	0	120

Accuracy:	0.686
Precision:	0.686
Recall:	1
F-measure:	0.814

Table 3: Confusion Matrix for Classification tree with default parameters

As we can see from the tree-above and the metrics, the model is clearly not functioning. The tree labels all new instances as patients, and will thus not have any practical value when used on new instances, even though its metrics is acceptable.

Task 3.3 In order to improve the performance of the classification tree, we are asked to tune some parameters. This can be done by either using a subset of attributes, change the maxDepth of the tree and so on. Observing that approximately one fourth of the training dataset is not-patients, I choose to allow the ctree to split variables with p-values higher than 0.05, by setting the mincriterion = 0.89, meaning that p is smaller than 0.11.

Using the library Boruta, I was able to find the most important features of the dataset. This was done using the function Boruta(Class .,data), which computes the importance of an attribute using permutation. The importance of each algorithm was then returned, ranked, and is summarized in the following table (Age apparently having highest importance):

Important Attributes:	Age, Alkphos, DB, Sgot, Sgpt, TB
Unresolved Attributes:	AG_Ratio, Albumin, Gender
Unimportant Attributes:	TP

The attributes I found to be yielding the best result was with Age, TB and Alkphos. I also increased the minbucket to 20, i.e. so that there are at least 20 instances in each node. Tuning other parameters didn't affect the tree, from what I saw at least - got the following tree anyhow:

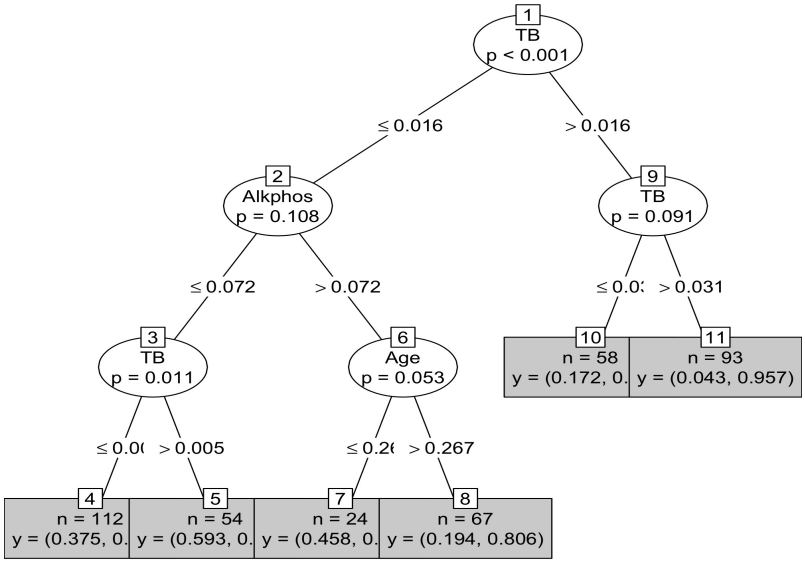


Figure 14: Ctree with tuned parameters

		Prediction	
		0	1
Class	0	11	44
	1	16	104

Accuracy:	0.657
Precision:	0.702
Recall:	0.867
F-measure:	0.776

Table 4: Confusion Matrix for Classification tree with default pararameters

Here, as we can see from the confusion matrix, the tree is at least more functional. Now, instead of only labeling every instance as Patient, the tree labels 11 out 55 Not-patients successfully. One disadvantage, critical in real-life, is that the classifier classifies 16/120 Patients as Not-patients. In other words, the classifier I ended with has a great need for improvement.

3.3 K-Nearest Neighbours

The K-Nearest-Neighbours algorithm is a classification algorithm based on feature similarity. In KNN, an object is classified based on its k-nearest neighbours using majority vote. For instance, using k=1 will only use the class of the closest neighbour. Hence, the selection of k influences the performance of the algorithm and the classification results; using a larger value for k can reduce the variance due to noise but can also tend to develop a bias so that other patterns can be ignored.

Task 3.4 I am going to use K-nearest neighbour with different values for k (2,3,4,5), and evaluate the classification results. In order to do this, I partition the data using the methods described in 3.1. Partitioning the data again is done to ensure the validity of the results. To perform the classification, I import the library class and use the function model = knn(train,test,cl,k), where train is the trainingdata, test is the testdata, cl is ground truth for trainingdata and k is number of neighbours considered. In order to evaluate the performance, i form a confusion matrix using the function table(test_gt,model), where test_gt is the ground truth for the test data.

		Prediction	
		0	1
Class	0	19	36
	1	36	84

Accuracy:	0.589
Precision:	0.700
Recall:	0.700
F-measure:	0.700

Table 5: Confusion Matrix for KNN with k = 1

		Prediction	
		0	1
Class	0	16	39
	1	27	93

Accuracy:	0.622
Precision:	0.775
Recall:	0.705
F-measure:	0.738

Table 6: Confusion Matrix for KNN with k = 2

		Prediction	
		0	1
Class	0	13	42
	1	25	95

Accuracy:	0.617
Precision:	0.792
Recall:	0.693
F-measure:	0.739

Table 7: Confusion Matrix for KNN with k = 3

		Prediction	
		0	1
Class	0	17	38
	1	20	100

Accuracy:	0.669
Precision:	0.833
Recall:	0.725
F-measure:	0.775

Table 8: Confusion Matrix for KNN with k = 4

		Prediction	
		0	1
Class	0	15	40
	1	16	104

Accuracy:	0.68
Precision:	0.867
Recall:	0.722
F-measure:	0.788

Table 9: Confusion Matrix for KNN with k = 5

We observe that KNN performs best on the dataset when k = 5. F-measure, which gives a harmonic combination of Recall and Precision, is highest for k = 5. In other words, for this dataset, it seems like choosing a higher value for k will help with the noise in the dataset and is therefore better.

References

[1] Mohamad, Usman. *Standardization and Its Effects on K-Means Clustering Algorithm.* https://tinyurl.com/y5e9kg5w/.

[2] Carlson, Gunnar. *Journal of Machine Learning Research 11 (2010), p.1447.* http://www.jmlr.org/papers/volume11/carlsson10a/carlsson10a.pdf.

[3] Erlandson, Erik. *Measuring Decision Tree Split Quality with Test Statistic P-Values.* http://erikerlandson.github.io/blog/2016/05/26/measuring-decision-tree-split-quality-with-test-statistic-p-values/.