Task1

Statistical Learning with Deep Artificial Neural Networks

Liv Breivik, Hannes Johansson, Alexander J Ohrt

13. april. 2022

The objective of this task is to use information on protein abundance and gene expression of patients to predict the breast invasive carcinoma (BRCA) estrogen receptor status.

Protein Abundance and Gene Expression Data Sets

```
gene.exp <- read_delim("gene_expression.csv", "\t", escape_double = FALSE, trim_ws = TRUE)
```

```
#> Rows: 526 Columns: 17815

#> -- Column specification --------------------------------------------------------------
#> Delimiter: "\t"
#> chr     (1): Sample
#> dbl (17814): ELMO2, CREB3L1, RPS11, PNMA1, MMP2, C10orf90, ZHX3, ERCC5, GPR98, R...

#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
prot.ab <- read_delim("protein_abundance.csv", "\t", escape_double = FALSE, trim_ws = TRUE)
```

```
#> Rows: 410 Columns: 143

#> -- Column specification --------------------------------------------------------------
#> Delimiter: "\t"
#> chr   (1): Sample
#> dbl (142): 14-3-3_epsilon, 4E-BP1, 4E-BP1_pS65, 4E-BP1_pT37, 4E-BP1_pT70, 53BP1,...

#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
clinical <- read_delim("clinical.csv","\t", escape_double = FALSE, trim_ws = TRUE)
```

```
#> Rows: 847 Columns: 19

#> -- Column specification --------------------------------------------------------------
#> Delimiter: "\t"
#> chr (19): Sample, Histology, PAM50Call, ajcc_cancer_metastasis_stage_code, ajcc_...

#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The dimensions of the protein abundance data set are shown below.

```r
dim(prot.ab)
```

```
#> [1] 410 143
```

```r
all(complete.cases(prot.ab)) # TRUE: There are no missing values.
```

```
#> [1] TRUE
```

The protein abundance data set has no missing values. It contains 410 unique patients, where each patient has their own sampling code and a numerical value that gives the abundance of 142 different proteins.

The dimensions of the gene expression data set are shown below.

```r
dim(gene.exp)
```

```
#> [1]   526 17815
```

```r
all(complete.cases(gene.exp)) # FALSE: There are missing values in some of the columns.
```

```
#> [1] FALSE
```

The gene expression data set has missing values. The missing values are simply removed from the data set in the following block of code.

```r
dim(gene.exp)
```

```
#> [1]   526 17815
```

```r
gene.exp2 <- gene.exp[complete.cases(gene.exp),]
dim(gene.exp2)
```

```
#> [1]   247 17815
```

Before the rows with missing values are removed, the gene expression data contains 526 unique patients, each of which has their own sampling code. After the rows with missing data are removed, the gene expression data contains 247 unique patients.

Next we find the patients with data of both types available, keeping in mind that the response we want to predict is contained in the `clinical` data. Before continuing, note that the `clinical` data has the following dimensions

```r
dim(clinical)
```

```
#> [1] 847  19
```

and we only care about columns 1 and 9. Column 1 contains the identifier of each patient, which are 847 in total in this data set. Column 9 contains the response we want to predict, which has the following unique values

```r
unique(clinical[,9])
```

```
#> # A tibble: 5 x 1
#>   breast_carcinoma_estrogen_receptor_status
#>   <chr>
#> 1 Positive
#> 2 Negative
#> 3 <NA>
#> 4 Not Performed
#> 5 Indeterminate
```

These values will be pre-processed later. Below the code used to find patients that have data available is given.

```r
full.gene.clin <- intersect(gene.exp2$Sample, clinical$Sample)
length(full.gene.clin)
```

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

The first intersection that is shown is the intersection between the gene expression data with no missing values and the clinical data, i.e. this intersection now contains all unique patient identifiers that exist in the gene data and that have recorded the response. Notice that this the data that will be used in the first part of the task.

```r
int.gene.prot <- intersect(gene.exp$Sample, prot.ab$Sample)
length(int.gene.prot)
```

```
#> [1] 404
```

The second intersection that is shown is the intersection between the gene expression data which still contains missing values and the protein abundance data. This will not be used in the analysis, but is given because it might be interesting to keep in mind. Thus we can see that 404 of the patients' sample codes exist in both the protein abundance and the gene expression data sets, before removing the rows with missing values from the gene expression data.

```r
int.gene.full.prot <- intersect(gene.exp2$Sample, prot.ab$Sample)
length(int.gene.full.prot)
```

```
#> [1] 190
```

The intersection above gives the unique patients that have no missing gene expression data and have recorded protein abundance data. This is used in order to define the next intersection.

```r
full.gene.prot.clin <- intersect(int.gene.full.prot, clinical$Sample)
length(full.gene.prot.clin)
```

```
#> [1] 181
```

The last intersection gives the intersection between the third list of patients (`int.gene.full.prot`) and the patients in the `clinical` data set. Thus, this contains the unique list of patients that have all necessary data in all three data sets. Notice that these patients will be used to define the complete data set, which will be used for the concatenated model later in the analysis (questions 7-10 in the task description).

As noted earlier, we will now only use the intersection `full.gene.clin` (to answer questions 2-6 in the task description). Even though we know that all these patients have a recorded breast invasive carcinoma (BRCA) estrogen receptor status, we need to check that the values they have recorded are either `Positive` or `Negative`, keeping in mind the values we saw that the `clinical` data set contains. Next, we remove all the individuals that don't have this information.

```r
chosen.data <- full.gene.clin

xclin <- clinical[,c(1,9)]
colnames(xclin) <- c("Sample", "BRCA")
xclin <- xclin[clinical$Sample %in% chosen.data, ]
xgene <- gene.exp2[gene.exp2$Sample %in% chosen.data, ]

sel1 <- which(xclin$BRCA != "Positive")
sel2 <- which(xclin$BRCA != "Negative")
sel <- intersect(sel1,sel2) # Find values of BRCA that are neither negative nor positive.
# In this case these values are either "Indeterminate", "Not Performed" or NA.
xclin <- xclin[-sel,] # Remove the rows with non-valid data for BRCA.
xclin <- xclin[-which(is.na(xclin$BRCA)),] # Also remove rows with missing data for BRCA.
```

```
# Join the (cleaned) clinical data and the gene expression data on "Sample".
mgene <- merge(xclin, xgene, by.x = "Sample", by.y = "Sample")
```

Gene Expression Data

Again, it is stressed that we now only use the **gene expression data**, i.e. the first mentioned set above (`full.gene.clin`). After the previous pre-process, this data set now contains the patients that have the complete gene expression data, as well as a well-defined BRCA receptor status.

Select the 25% of genes with the most variability.

The 25% percent of genes with the most variability are chosen. Information about the genes chosen are stored and reused later in the selection of the genes for the complete data set in section 4 and onwards, to make sure that the same set of genes (features) that the SAE was trained with are the ones selected in the complete data set as well.

```
percentage <- round(dim(mgene[,-c(1,2)])[[2]]*0.25) # Find how many variables correspond to 25%.
variances <- apply(X=mgene[,-c(1,2)], MARGIN=2, FUN=var) # Find empirical variance in each of the varia
sorted <- sort(variances, decreasing=TRUE, index.return=TRUE)$ix[1:percentage] # Sort from highest to l
mgene.lvar <- mgene[, c(1,2,sorted)] # Select the 25% largest variance variables using the indices foun
```

The selected 4454 genes are used to implement a stacked autoencoder (SAE) with three stacked layers of 1000, 100 and 50 nodes.

Final Training/Test Split

```
set.seed(params$seed)
training.fraction <- 0.70 # 70 % of data will be used for training.
training <- sample(1:nrow(mgene.lvar),nrow(mgene.lvar)*training.fraction)

xtrain <- mgene.lvar[training,-c(1,2)]
xtest <- mgene.lvar[-training,-c(1,2)]

# Scaling for better numerical stability.
# This is a standard "subtract mean and divide by standard deviation" scaling.
xtrain <- scale(data.matrix(xtrain))
xtest <- scale(data.matrix(xtest))

# Pick out labels for train and test set.
ytrain <- mgene.lvar[training,2]
ytest <- mgene.lvar[-training,2]

# Change labels to numerical values in train and test set.
ylabels <- c()
ylabels[ytrain=="Positive"] <- 1
ylabels[ytrain=="Negative"] <- 0

ytestlabels <- c()
ytestlabels[ytest=="Positive"] <- 1
ytestlabels[ytest=="Negative"] <- 0

# The data is saved to a file, so that it can be loaded directly into tfruns() files.
data.train <- data.frame(ylabels, xtrain)
data.test <- data.frame(ytestlabels, xtest)
```

```
write.csv(data.train, "train_for5.csv")
write.csv(data.test, "test_for5.csv")
```

Implementation of SAE

In this section a stacked autoencoder (SAE) will be implemented. It will consist of three stacked layers of 1000, 100 and 50 nodes. In each case, some qualitative evidence of the quality of coding obtained will be given, in the form of correlation plots between input and output.

First Layer (1000 nodes)

```
# Develop the encoder.
input_enc1 <- layer_input(shape = percentage)
output_enc1 <- input_enc1 %>%
  layer_dense(units=1000,activation="relu")
encoder1 <- keras_model(input_enc1, output_enc1)
summary(encoder1)
```

```
#> Model: "model_51"
#> __________________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ==========================================================================================================
#>  input_37 (InputLayer)                                       [(None, 4454)]
#>
#>  dense_63 (Dense)                                            (None, 1000)
#>
#> ==========================================================================================================
#> Total params: 4,455,000
#> Trainable params: 4,455,000
#> Non-trainable params: 0
#> __________________________________________________________________________________________________________
```

```
# Develop the decoder.
input_dec1 <- layer_input(shape = 1000)
output_dec1 <- input_dec1 %>%
  layer_dense(units = percentage, activation="linear")
decoder1 <- keras_model(input_dec1, output_dec1)
summary(decoder1)
```

```
#> Model: "model_52"
#> __________________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ==========================================================================================================
#>  input_38 (InputLayer)                                       [(None, 1000)]
#>
#>  dense_64 (Dense)                                            (None, 4454)
#>
#> ==========================================================================================================
#> Total params: 4,458,454
#> Trainable params: 4,458,454
#> Non-trainable params: 0
#> __________________________________________________________________________________________________________
```

```
# Develop the first AE.
aen_input1 <- layer_input(shape = percentage)
aen_output1 <- aen_input1 %>%
  encoder1() %>%
  decoder1()
sae1 <- keras_model(aen_input1, aen_output1)
summary(sae1)
```

```
#> Model: "model_53"
#> ____________________________________________________________________________________________________________
#>  Layer (type)                                               Output Shape
#> ============================================================================================================
#>  input_39 (InputLayer)                                      [(None, 4454)]
#> 
#>  model_51 (Functional)                                      (None, 1000)
#> 
#>  model_52 (Functional)                                      (None, 4454)
#> 
#> ============================================================================================================
#> Total params: 8,913,454
#> Trainable params: 8,913,454
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________________
```

We compile the model and fit it to the training data. To decide the final number of epochs each 'val-loss'-value is regarded after each epoch. If the value has not decreased in a certain number of epochs the training will stop. This is to reduce the risk of overfitting the network, that is that the network may otherwise learn patterns that are too specific for the training data while the performance on the actual validation data begins to decrease.

```
sae1 %>% compile(
  optimizer = "rmsprop",
  loss = "mse"
)
```

```
callbacks_parameters <- callback_early_stopping(
  monitor = "val_loss",
  patience = 12,
  verbose = 1,
  mode = "min",
  restore_best_weights = FALSE
)

sae1 %>% fit(
  x = xtrain,
  y = xtrain,
  epochs = 40,
  batch_size = 64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

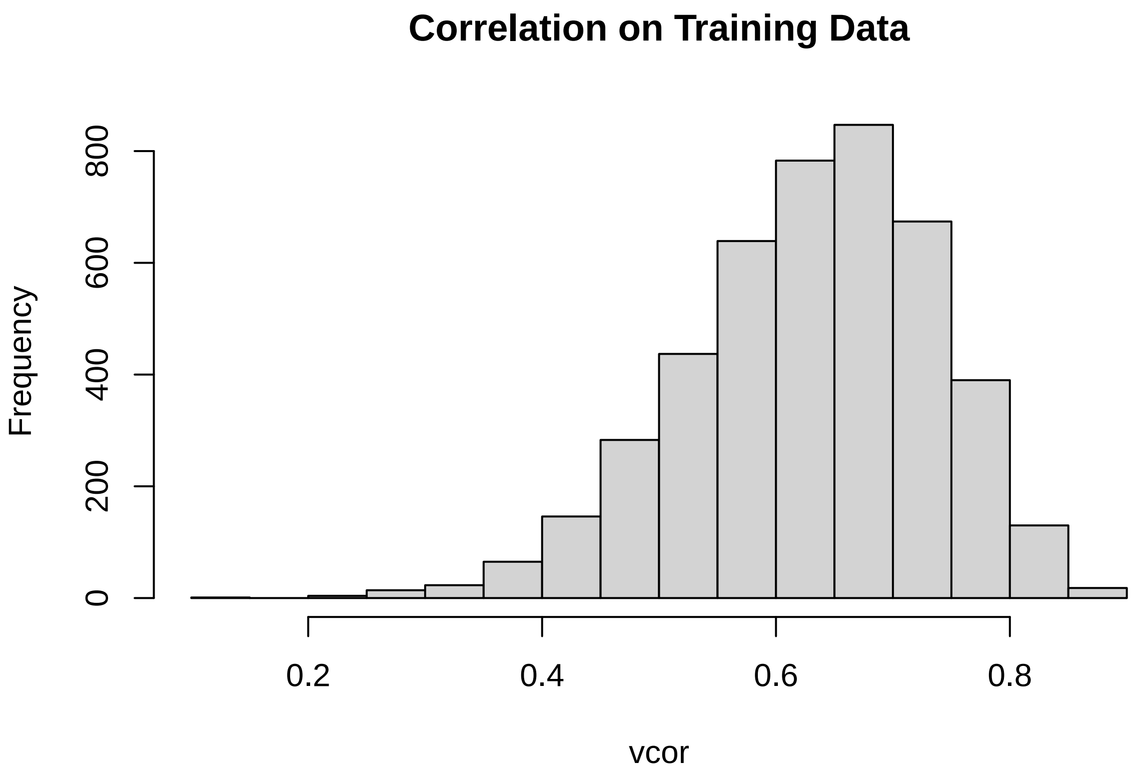
We make predictions on the training data.

```
encoded_expression1 <- encoder1 %>% predict(xtrain)
decoded_expression1 <- decoder1 %>% predict(encoded_expression1)
```

```r
# This method gives the same predictions as the two lines above.
# i.e. the values in decoded_expression above are the same as the values in x.hat below.
x.hat <- predict(sae1,xtrain)
```

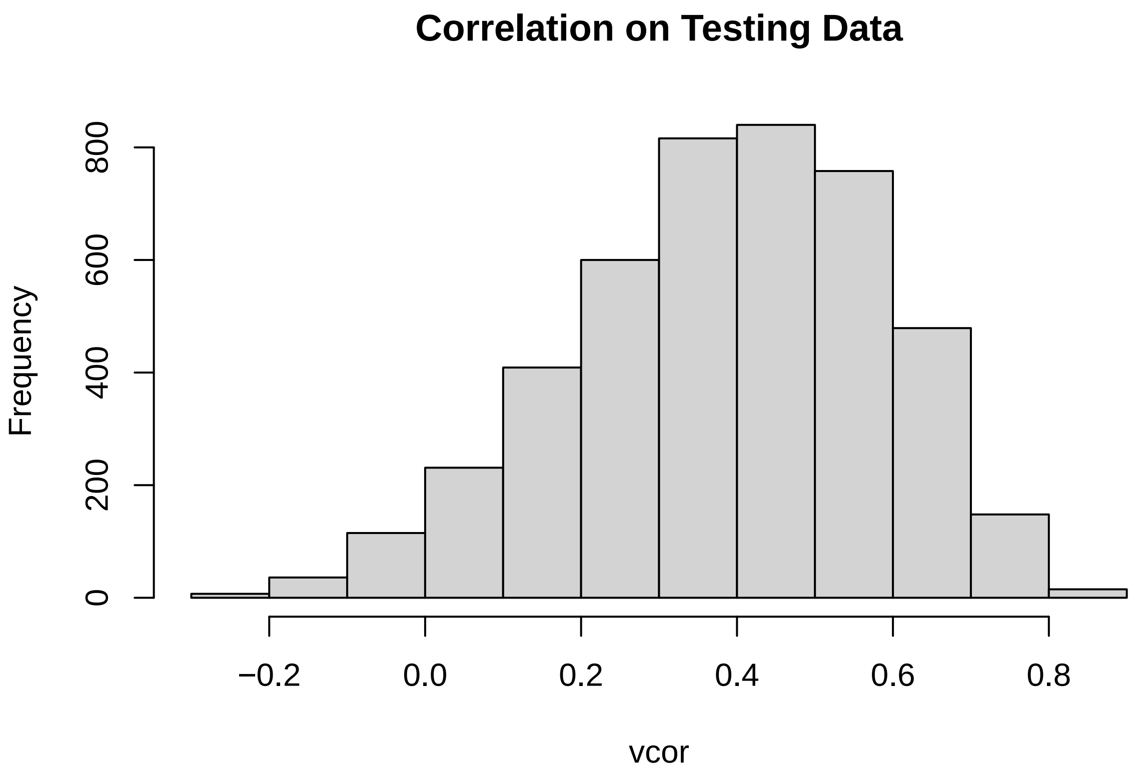
Some (weak) evidence of the quality of the coding obtained follows. We plot the correlation between the predictions from the autoencoder and the correct data, both on the train and test sets. The results are shown below.

```r
vcor <- diag(cor(x.hat,xtrain))
hist(vcor, main = "Correlation on Training Data")
```



The histogram above shows that the correlation between the training data and the predictions from the first autoencoder are relatively high. Worth mentioning is that if we had not implemented the training stop when no progression of the validation data was registered, the correlation on the training set would be significantly higher, as the network would continue to fit the parameters to the training data. As stated, this would probably have resulted in worse values for the initial test set that we set aside before the training started. We do the same check on the test set.

```r
x.hat <- predict(sae1,xtest)
vcor <- diag(cor(x.hat,xtest))
hist(vcor, main = "Correlation on Testing Data")
```



As expected, the correlation is lower on the test data, but there still is some correlation.

Second Layer (100 nodes)

```
# Develop the encoder.
input_enc2 <- layer_input(shape = 1000)
output_enc2 <- input_enc2 %>%
  layer_dense(units=100,activation="relu")
encoder2 <- keras_model(input_enc2, output_enc2)
summary(encoder2)
```

```
#> Model: "model_54"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ====================================================================================================
#>  input_40 (InputLayer)                                   [(None, 1000)]
#>
#>  dense_65 (Dense)                                        (None, 100)
#>
#> ====================================================================================================
#> Total params: 100,100
#> Trainable params: 100,100
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________
```

```
# Develop the decoder.
input_dec2 <- layer_input(shape = 100)
output_dec2 <- input_dec2 %>%
  layer_dense(units = 1000, activation="linear")
decoder2 <- keras_model(input_dec2, output_dec2)
```

```
summary(decoder2)
```

```
#> Model: "model_55"
#> ____________________________________________________________________________________
#>  Layer (type)                                                   Output Shape
#> ====================================================================================
#>  input_41 (InputLayer)                                          [(None, 100)]
#>
#>  dense_66 (Dense)                                               (None, 1000)
#>
#> ====================================================================================
#> Total params: 101,000
#> Trainable params: 101,000
#> Non-trainable params: 0
#> ____________________________________________________________________________________
```

```
# Develop the second AE.
aen_input2 <- layer_input(shape = 1000)
aen_output2 <- aen_input2 %>%
  encoder2() %>%
  decoder2()
sae2 <- keras_model(aen_input2, aen_output2)
summary(sae2)
```

```
#> Model: "model_56"
#> ____________________________________________________________________________________
#>  Layer (type)                                                   Output Shape
#> ====================================================================================
#>  input_42 (InputLayer)                                          [(None, 1000)]
#>
#>  model_54 (Functional)                                          (None, 100)
#>
#>  model_55 (Functional)                                          (None, 1000)
#>
#> ====================================================================================
#> Total params: 201,100
#> Trainable params: 201,100
#> Non-trainable params: 0
#> ____________________________________________________________________________________
```

We compile the model and fit it to the training data.

```
sae2 %>% compile(
  optimizer = "rmsprop",
  loss = "mse"
)
```

```
callbacks_parameters <- callback_early_stopping(
  monitor = "val_loss",
  patience = 8,
  verbose = 1,
  mode = "min",
  restore_best_weights = FALSE
)

sae2 %>% fit(
  x = encoded_expression1,
  y = encoded_expression1,
  epochs = 100,
  batch_size = 64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

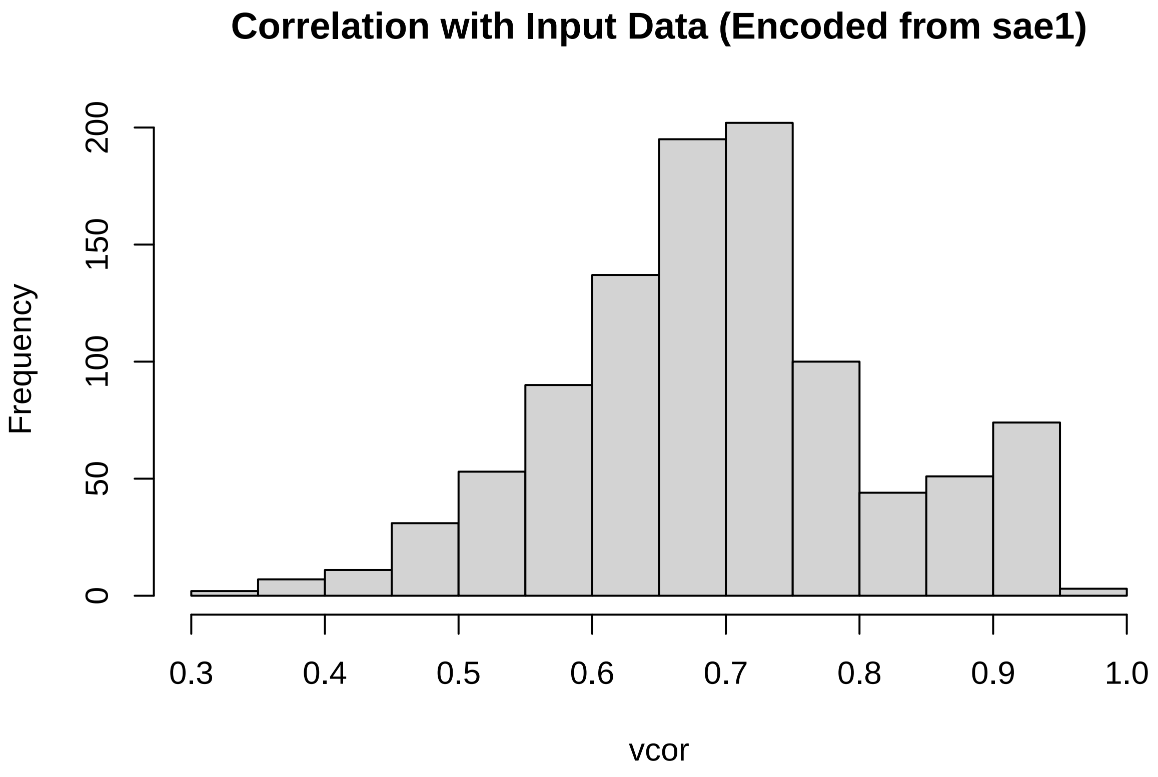
We make predictions on the training data, which in this case is the training data reduced in dimension from the first autoencoder. Also here we use the automatic training stop when the performance on the validation set has stopped improving.

```
encoded_expression2 <- encoder2 %>% predict(encoded_expression1)
decoded_expression2 <- decoder2 %>% predict(encoded_expression2)

# This method gives the same predictions as the two lines above.
# i.e. the values in decoded_expression above are the same as the values in x.hat below.
x.hat <- predict(sae2,encoded_expression1)
```

Some (weak) evidence of the quality of the coding obtained follows. Now we plot the correlation between the predictions from the autoencoder and the input data, which in this case is the encoded data from the first autoencoder (the latent variables/space).

```
vcor <- diag(cor(x.hat,encoded_expression1))
hist(vcor, main = "Correlation with Input Data (Encoded from sae1)")
```



The histogram above shows that there is correlation between the input and the predictions from the second autoencoder. Note that we do not have a test set in this case, since the dimension of the output data from sae2 is 1000, which is less than the amount of features in the test data.

Third Layer (50 nodes)

```r
# Develop the encoder.
input_enc3 <- layer_input(shape = 100)
output_enc3 <- input_enc3 %>%
  layer_dense(units=50,activation="relu")
encoder3 <- keras_model(input_enc3, output_enc3)
summary(encoder3)
```

```
#> Model: "model_57"
#> ______________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ==============================================================================================
#>  input_43 (InputLayer)                                       [(None, 100)]
#>
#>  dense_67 (Dense)                                            (None, 50)
#>
#> ==============================================================================================
#> Total params: 5,050
#> Trainable params: 5,050
#> Non-trainable params: 0
#> ______________________________________________________________________________________________
```

```r
# Develop the decoder.
input_dec3 <- layer_input(shape = 50)
output_dec3 <- input_dec3 %>%
  layer_dense(units = 100, activation="linear")
decoder3 <- keras_model(input_dec3, output_dec3)
summary(decoder3)
```

```
#> Model: "model_58"
#> ______________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ==============================================================================================
#>  input_44 (InputLayer)                                       [(None, 50)]
#>
#>  dense_68 (Dense)                                            (None, 100)
#>
#> ==============================================================================================
#> Total params: 5,100
#> Trainable params: 5,100
#> Non-trainable params: 0
#> ______________________________________________________________________________________________
```

```r
# Develop the third AE.
aen_input3 <- layer_input(shape = 100)
aen_output3 <- aen_input3 %>%
  encoder3() %>%
  decoder3()
sae3 <- keras_model(aen_input3, aen_output3)
summary(sae3)
```

```
#> Model: "model_59"
#> ______________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ==============================================================================================
#>  input_45 (InputLayer)                                   [(None, 100)]
#>
#>  model_57 (Functional)                                   (None, 50)
#>
#>  model_58 (Functional)                                   (None, 100)
#>
#> ===========================================================================================
#> Total params: 10,150
#> Trainable params: 10,150
#> Non-trainable params: 0
#> ___________________________________________________________________________________________
```

We compile the model and fit it to the training data.

```r
sae3 %>% compile(
  optimizer = "rmsprop",
  loss = "mse"
)
```

```r
sae3 %>% fit(
  x = encoded_expression2,
  y = encoded_expression2,
  epochs = 180,
  batch_size = 64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

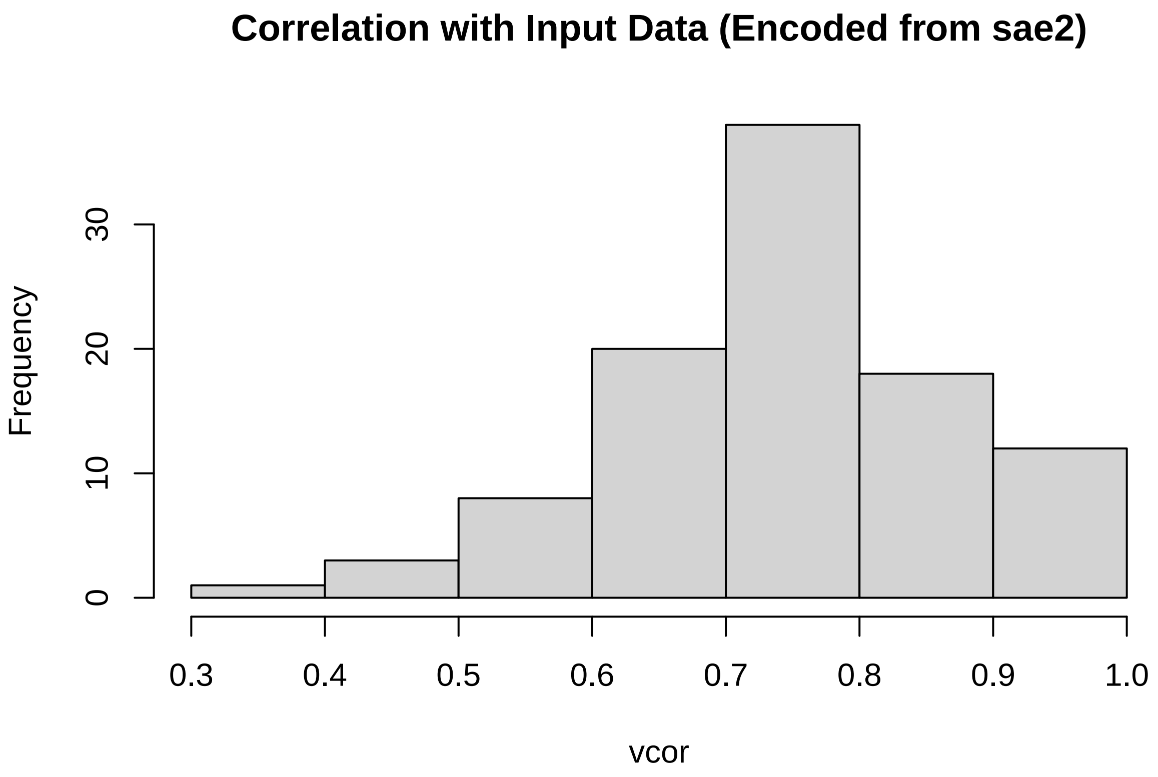
We make predictions on the training data, which in this case is the training data reduced in dimension from the first autoencoder.

```r
encoded_expression3 <- encoder3 %>% predict(encoded_expression2)
decoded_expression3 <- decoder3 %>% predict(encoded_expression3)

# This method gives the same predictions as the two lines above.
# i.e. the values in decoded_expression above are the same as the values in x.hat below.
x.hat <- predict(sae3,encoded_expression2)
```

Some (weak) evidence of the quality of the coding obtained follows. Now we plot the correlation between the predictions from the autoencoder and the input data, which in this case is the encoded data from the first autoencoder (the latent variables/space).

```r
vcor <- diag(cor(x.hat,encoded_expression2))
hist(vcor, main = "Correlation with Input Data (Encoded from sae2)")
```



The histogram above shows that there is some correlation between the input and the predictions from the second autoencoder. Notice that the correlation is less than earlier though. Also note that we do not have a test set in this case, since the dimension of the output data from sae3 is 100, which is less than the amount of features in the test data.

Final Model (SAE)

The final stacked autoencoder (SAE) is constructed below. All the encoders previously trained are stacked together.

```
sae_input <- layer_input(shape = percentage, name = "gene.mod")
sae_output <- sae_input %>%
  encoder1() %>%
  encoder2() %>%
  encoder3()

sae <- keras_model(sae_input, sae_output)
summary(sae)
```

```
#> Model: "model_60"
#> ____________________________________________________________________________________________________________
#>  Layer (type)                                                  Output Shape
#> ============================================================================================================
#>  gene.mod (InputLayer)                                         [(None, 4454)]
#>
#>  model_51 (Functional)                                         (None, 1000)
#>
#>  model_54 (Functional)                                         (None, 100)
#>
#>  model_57 (Functional)                                         (None, 50)
#>
```

```
#> ================================================================================================
#> Total params: 4,560,150
#> Trainable params: 4,560,150
#> Non-trainable params: 0
#> ________________________________________________________________________________________________
```

```
# Code below is used for loading model (and model checking) in separate file for tfruns().
yhat <- predict(sae, xtest)
write.csv(yhat, file = "predictions.csv")
save_model_weights_hdf5(sae, "sae.hdf5")
```

SAE as Pre-Training Model for Prediction of Estrogen Receptor State

The SAE is used as a pre-training model for prediction of the estrogen receptor state. The DNN has 10 nodes in the first layer, followed by one output node. The weights are frozen for the first 3 functional layers, which means that only the weights from the third autoencoder to the first fully connected layer and from the first layer in the DNN to the output layer (in total 521 weights) NOTE TO SELF: GJØR DETTE TALLET DYNAMISK OGSÅ (HVIS MULIG) are to be fine-tuned to obtain the final classifier.

```
sae_output2 <- sae_output %>%
  layer_dense(10,activation = "relu") %>% # Couple with fully connected layers (DNN).
  layer_dense(1,activation = "sigmoid")

sae <- keras_model(sae_input, sae_output2)
summary(sae)
```

```
#> Model: "model_61"
#> ________________________________________________________________________________________________
#>  Layer (type)                                        Output Shape
#> ================================================================================================
#>  gene.mod (InputLayer)                               [(None, 4454)]
#>
#>  model_51 (Functional)                               (None, 1000)
#>
#>  model_54 (Functional)                               (None, 100)
#>
#>  model_57 (Functional)                               (None, 50)
#>
#>  dense_70 (Dense)                                    (None, 10)
#>
#>  dense_69 (Dense)                                    (None, 1)
#>
#> ================================================================================================
#> Total params: 4,560,671
#> Trainable params: 4,560,671
#> Non-trainable params: 0
#> ________________________________________________________________________________________________
```

```
freeze_weights(sae,from=1,to=4) # Freeze the weights (pre-training using the SAE).
# Only the weights in the two coupled fully connected layers will be trained when freezed in this manne
summary(sae)
```

```
#> Model: "model_61"
#> ________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ===================================================================================================
#>  gene.mod (InputLayer)                                   [(None, 4454)]
#>
#>  model_51 (Functional)                                   (None, 1000)
#>
#>  model_54 (Functional)                                   (None, 100)
#>
#>  model_57 (Functional)                                   (None, 50)
#>
#>  dense_70 (Dense)                                        (None, 10)
#>
#>  dense_69 (Dense)                                        (None, 1)
#>
#> ===================================================================================================
#> Total params: 4,560,671
#> Trainable params: 521
#> Non-trainable params: 4,560,150
#> ___________________________________________________________________________________________________
```

We compile and fit the final classifier.

```
sae %>% compile(
  optimizer = "rmsprop",
  loss = 'binary_crossentropy',
  metric = "acc"
)
```

```
sae %>% fit(
  x=xtrain,
  y=ylabels,
  epochs = 120,
  batch_size=64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

Performance Metrics

The model is evaluated on the test set below.

```
sae %>%
  evaluate(xtest, ytestlabels)
```

```
#>      loss       acc
#> 0.2660313 0.9285714
```

Predictions on the test set are calculated. The classifier is built on the assumption that predictions of probability smaller than 0.5 are negative receptor states, while probabilities larger than 0.5 are positive receptor states. The confusion matrix of the predictions is shown below.

```
yhat <- predict(sae,xtest)
yhatclass<-as.factor(ifelse(yhat<0.5,0,1))
confusionMatrix(yhatclass,as.factor(ytestlabels))
```

```
#> Confusion Matrix and Statistics
#>
```

```
#>           Reference
#> Prediction  0  1
#>          0 17  3
#>          1  2 48
#>
#>                Accuracy : 0.9286
#>                  95% CI : (0.8411, 0.9764)
#>     No Information Rate : 0.7286
#>     P-Value [Acc > NIR] : 2.543e-05
#>
#>                   Kappa : 0.8223
#>
#>  Mcnemar's Test P-Value : 1
#>
#>             Sensitivity : 0.8947
#>             Specificity : 0.9412
#>          Pos Pred Value : 0.8500
#>          Neg Pred Value : 0.9600
#>              Prevalence : 0.2714
#>          Detection Rate : 0.2429
#>    Detection Prevalence : 0.2857
#>       Balanced Accuracy : 0.9180
#>
#>        'Positive' Class : 0
#>
```

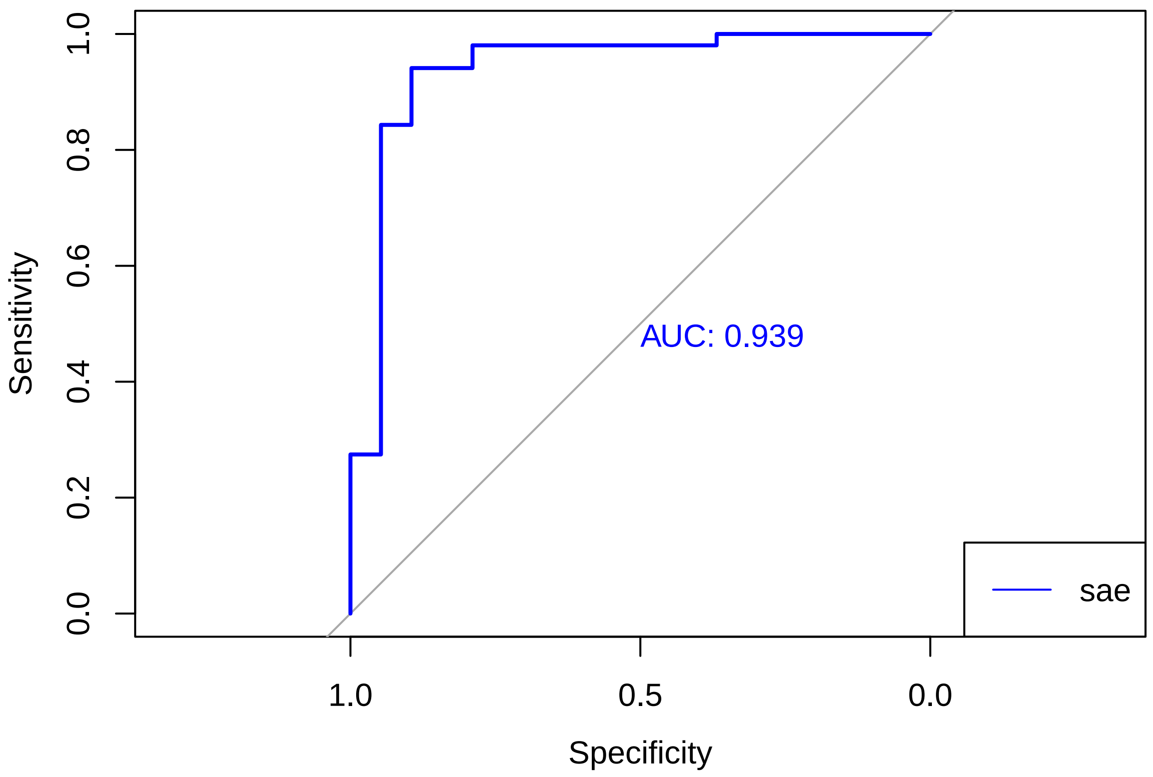
The ROC curve is shown below.

```
roc_sae_test <- roc(response = ytestlabels, predictor = as.numeric(yhat))
```

```
#> Setting levels: control = 0, case = 1
```

```
#> Setting direction: controls < cases
```

```
plot(roc_sae_test, col = "blue", print.auc=TRUE)
legend("bottomright", legend = c("sae"), lty = c(1), col = c("blue"))
```



As is seen above, the AUC is 0.94.

For an AUC curve, a score of 1 would signify a perfect predictor whilst a score of 0.5 would signify that the predictor is just as good as a random generator (i.e. equivalent to flipping a coin). A score of 0.94 is therefore considered acceptable as it should be able to distinguish the various cases to some extent, however there is big room for improvement.

Use `tfruns` to Explore Configurations of First Fully Connected Layer

In this section, `tfruns` is used to explore what amount of nodes in the first layer of the DNN gives the best performance. We are asked to search among three different numbers; 5, 10 and 20. The code used with `tfruns` is given in separate R files. Notice that we perhaps could have sourced the code into this file, like will be done later with the concatenated model, but in this case we preferred the option of running the file separately while developing the models. The exploration leads to the conclusion that the configuration with 20 nodes gives the best results.

Thus, the final model is

```
sae_output2 <- sae_output %>%
  layer_dense(20,activation = "relu") %>% # Couple with fully connected layers (DNN).
  layer_dense(1,activation = "sigmoid")

sae <- keras_model(sae_input, sae_output2)
summary(sae)
```

```
#> Model: "model_62"
#> ________________________________________________________________________________________________________
#>  Layer (type)                                                     Output Shape
#> ========================================================================================================
#>  gene.mod (InputLayer)                                   [(None, 4454)]
#>
#>  model_51 (Functional)                                   (None, 1000)
#>
#>  model_54 (Functional)                                   (None, 100)
#>
#>  model_57 (Functional)                                   (None, 50)
#>
#>  dense_72 (Dense)                                        (None, 20)
#>
#>  dense_71 (Dense)                                        (None, 1)
#>
#> ====================================================================================================
#> Total params: 4,561,191
#> Trainable params: 1,041
#> Non-trainable params: 4,560,150
#> ____________________________________________________________________________________________________
```

```r
freeze_weights(sae,from=1,to=4) # Freeze the weights (pre-training using the SAE).
summary(sae)
```

```
#> Model: "model_62"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ====================================================================================================
#>  gene.mod (InputLayer)                                   [(None, 4454)]
#>
#>  model_51 (Functional)                                   (None, 1000)
#>
#>  model_54 (Functional)                                   (None, 100)
#>
#>  model_57 (Functional)                                   (None, 50)
#>
#>  dense_72 (Dense)                                        (None, 20)
#>
#>  dense_71 (Dense)                                        (None, 1)
#>
#> ====================================================================================================
#> Total params: 4,561,191
#> Trainable params: 1,041
#> Non-trainable params: 4,560,150
#> ____________________________________________________________________________________________________
```

```r
sae %>% compile(
  optimizer = "rmsprop",
  loss = 'binary_crossentropy',
  metric = "acc"
)

sae %>% fit(
  x=xtrain,
  y=ylabels,
  epochs = 80,
  batch_size=64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)

sae %>%
  evaluate(xtest, ytestlabels)
```

```
#>      loss       acc
#> 0.2756056 0.9000000
```

```
yhat <- predict(sae,xtest)
yhatclass<-as.factor(ifelse(yhat<0.5,0,1))
confusionMatrix(yhatclass,as.factor(ytestlabels))
```

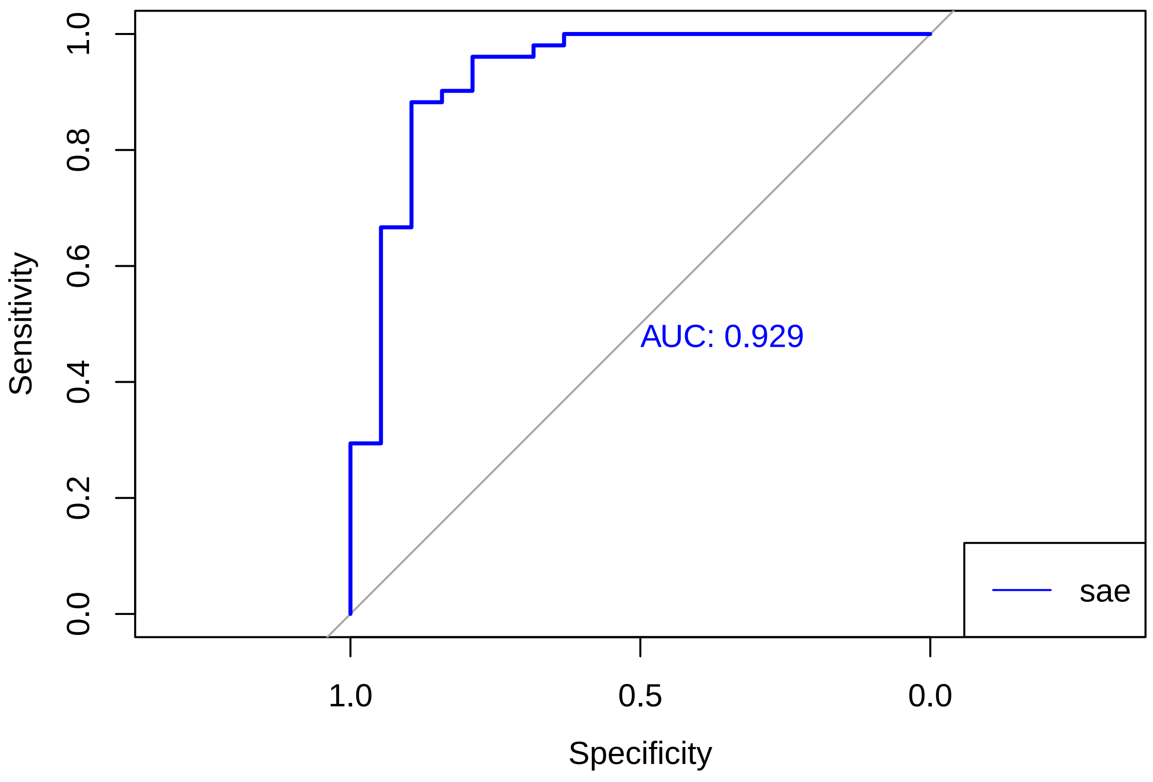
```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  0  1
#>          0 15  3
#>          1  4 48
#>
#>                Accuracy : 0.9
#>                  95% CI : (0.8048, 0.9588)
#>     No Information Rate : 0.7286
#>     P-Value [Acc > NIR] : 0.0003901
#>
#>                   Kappa : 0.7429
#>
#>  Mcnemar's Test P-Value : 1.0000000
#>
#>             Sensitivity : 0.7895
#>             Specificity : 0.9412
#>          Pos Pred Value : 0.8333
#>          Neg Pred Value : 0.9231
#>              Prevalence : 0.2714
#>          Detection Rate : 0.2143
#>    Detection Prevalence : 0.2571
#>       Balanced Accuracy : 0.8653
#>
#>        'Positive' Class : 0
#>
```

```
roc_sae_test <- roc(response = ytestlabels, predictor = as.numeric(yhat))
```

```
#> Setting levels: control = 0, case = 1
```

```
#> Setting direction: controls < cases
```

```
plot(roc_sae_test, col = "blue", print.auc=TRUE)
legend("bottomright", legend = c("sae"), lty = c(1), col = c("blue"))
```



From the ROC curve we see that the AUC-score now is closer to 1, indicating that this is a better model.

Complete Data

The **complete data** (gene expression and protein abundance) is split into train and test sets.

As explained in section 1, the same set of gene features is used in this data set, in order to secure that the input now consists of the same genes that the network was trained on in previous sections.

```
chosen.data <- full.gene.prot.clin # Complete data set, as defined in part 1.
xclin <- clinical[,c(1,9)]
colnames(xclin) <- c("Sample", "BRCA")
xclin <- xclin[clinical$Sample %in% chosen.data, ]
xprot <- prot.ab[prot.ab$Sample%in%chosen.data,]
xgene <- gene.exp2[gene.exp2$Sample %in% chosen.data, ]

sel1 <- which(xclin$BRCA != "Positive")
sel2 <- which(xclin$BRCA != "Negative")
sel <- intersect(sel1,sel2) # Find values of BRCA that are neither negative nor positive.
# In this case these values are either "Indeterminate", "Not Performed" or NA.
xclin <- xclin[-sel,] # Remove the rows with non-valid data for BRCA.
xclin <- xclin[-which(is.na(xclin$BRCA)),] # Also remove rows with missing data for BRCA.

# Join the (cleaned) clinical data and the gene expression data on "Sample".
mgene <- merge(xclin, xgene, by.x = "Sample", by.y = "Sample")
mtot <- merge(mgene, xprot, by.x = "Sample", by.y = "Sample")
#mtot <- xclin %>% left_join(xgene) %>% left_join(xprot, by = "Sample") # This gives the same result.

mtot.patients <- mtot[,1] # Select patient sample codes that have all data available.

set.seed(params$seed)
```

```
training.fraction <- 0.70 # 70 % of data will be used for training.
training <- sample(1:nrow(mtot),nrow(mtot)*training.fraction)

xprot2 <- xprot %>% dplyr::filter(Sample %in% mtot.patients)
xprot.train <- xprot2[training,-1]
xprot.test <- xprot2[-training,-1]

xgene2 <- xgene %>% dplyr::filter(Sample %in% mtot.patients)

# Pick out the same features as in the first model (as noted in section 1.1.1).
xgene2 <- xgene2[, c(1,sorted)] # Select the 25% largest variance variables using the indices found in
# We also select the patient sample id, following the same pattern as above.
xgene.train <- xgene2[training,-1]
xgene.test <- xgene2[-training,-1]

# Scaling for better numerical stability.
# This is a standard "subtract mean and divide by standard deviation" scaling.
xprot.test <- scale(data.matrix(xprot.test))
xprot.train <- scale(data.matrix(xprot.train))
xgene.test <- scale(data.matrix(xgene.test))
xgene.train <- scale(data.matrix(xgene.train))

# ADDITIONAL CHECK: Make sure that the patients of the labels and feature data are the same.
all.equal(tibble(mtot[,1]), xgene2[,1], check.attributes = F)
```

```
#> [1] TRUE
```

```
all.equal(tibble(mtot[,1]), xprot2[,1], check.attributes = F)
```

```
#> [1] TRUE
```

```
ytrain <- mtot[training,2] # Pick out training labels.
ytest <- mtot[-training,2] # Pick out testing labels.

# Change labels to numerical values in train and test set.
ylabels.comp <- c()
ylabels.comp[ytrain=="Positive"] <- 1
ylabels.comp[ytrain=="Negative"] <- 0

ytestlabels.comp <- c()
ytestlabels.comp[ytest=="Positive"] <- 1
ytestlabels.comp[ytest=="Negative"] <- 0

# Lables are change to categorical, in order to facilitate two output nodes in concatenated model.
ytrain.bin <- to_categorical(as.array(ylabels.comp), 2)
ytest.bin <- to_categorical(as.array(ytestlabels.comp), 2)
```

Importing the SAE from the class example.

The SAE for the abundance of proteins (from class examples) is added in the code block below. Then, this model is concatenated with the former model, that was trained on the gene expression data.

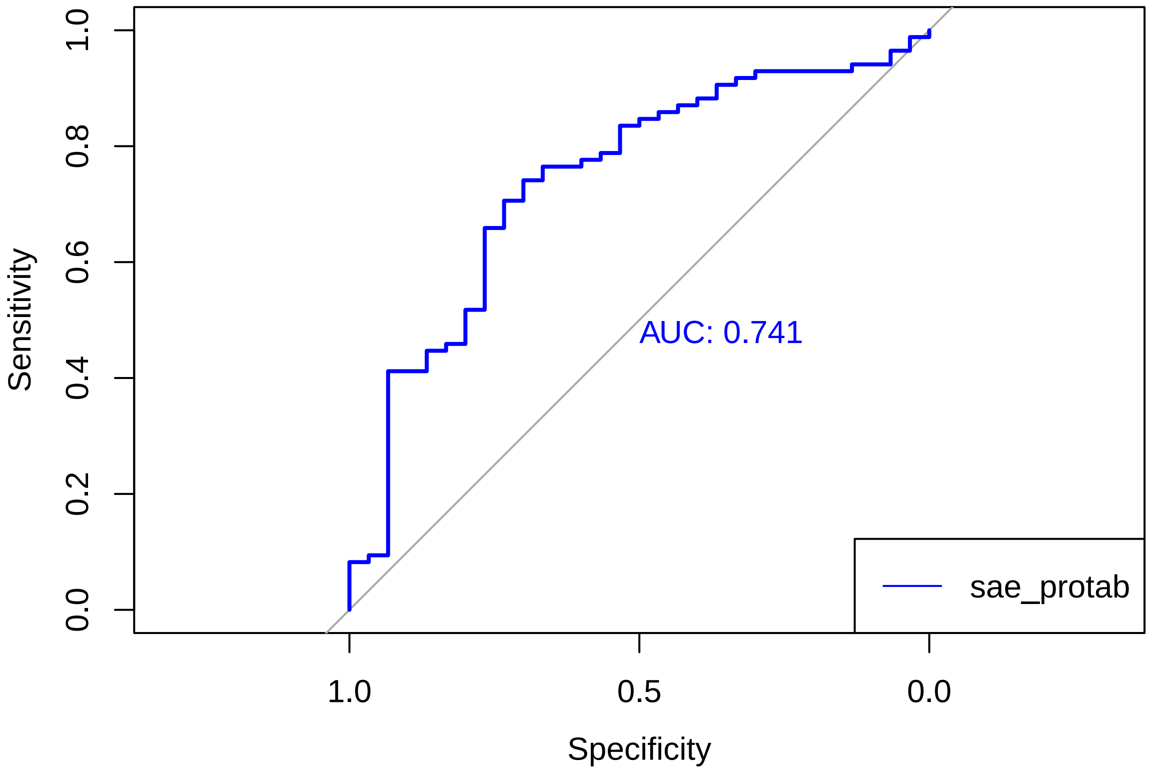
```
# Model given by teaching staff. A lot of code from the original file is removed and variables are rena
source("aes_practice_3.R") # source: Parses the code and evaluates it in this environment.
```

```
#> Model: "model_63"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ====================================================================================================
#>  input_46 (InputLayer)                                       [(None, 142)]
#>
#>  dense_73 (Dense)                                            (None, 50)
#>
#> ====================================================================================================
#> Total params: 7,150
#> Trainable params: 7,150
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________
#> Model: "model_64"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ====================================================================================================
#>  input_47 (InputLayer)                                       [(None, 50)]
#>
#>  dense_74 (Dense)                                            (None, 142)
#>
#> ====================================================================================================
#> Total params: 7,242
#> Trainable params: 7,242
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________
#> Model: "model_65"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ====================================================================================================
#>  input_48 (InputLayer)                                       [(None, 142)]
#>
#>  model_63 (Functional)                                       (None, 50)
#>
#>  model_64 (Functional)                                       (None, 142)
#>
#> ====================================================================================================
#> Total params: 14,392
#> Trainable params: 14,392
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________
#> Model: "model_66"
#> ____________________________________________________________________________________________________
#>  Layer (type)                                                Output Shape
#> ====================================================================================================
#>  input_49 (InputLayer)                                       [(None, 50)]
#>
#>  dense_75 (Dense)                                            (None, 20)
#>
#> ====================================================================================================
#> Total params: 1,020
#> Trainable params: 1,020
#> Non-trainable params: 0
#> ____________________________________________________________________________________________________
#> Model: "model_67"
#> __________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ==================================================================================================
#>  input_50 (InputLayer)                                   [(None, 20)]
#>
#>  dense_76 (Dense)                                        (None, 50)
#>
#> ==================================================================================================
#> Total params: 1,050
#> Trainable params: 1,050
#> Non-trainable params: 0
#> __________________________________________________________________________________________________
#> Model: "model_68"
#> __________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ==================================================================================================
#>  input_51 (InputLayer)                                   [(None, 50)]
#>
#>  model_66 (Functional)                                   (None, 20)
#>
#>  model_67 (Functional)                                   (None, 50)
#>
#> ==================================================================================================
#> Total params: 2,070
#> Trainable params: 2,070
#> Non-trainable params: 0
#> __________________________________________________________________________________________________
#> Model: "model_69"
#> __________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ==================================================================================================
#>  input_52 (InputLayer)                                   [(None, 20)]
#>
#>  dense_77 (Dense)                                        (None, 10)
#>
#> ==================================================================================================
#> Total params: 210
#> Trainable params: 210
#> Non-trainable params: 0
#> __________________________________________________________________________________________________
#> Model: "model_70"
#> __________________________________________________________________________________________________
#>  Layer (type)                                            Output Shape
#> ==================================================================================================
#>  input_53 (InputLayer)                                   [(None, 10)]
#>
#>  dense_78 (Dense)                                        (None, 20)
#>
#> ==================================================================================================
#> Total params: 220
#> Trainable params: 220
#> Non-trainable params: 0
#> __________________________________________________________________________________________________
#> Model: "model_71"
#> ____________________________________________________________________________________________
#>  Layer (type)                                                    Output Shape
#> ============================================================================================
#>  input_54 (InputLayer)                                           [(None, 20)]
#>
#>  model_69 (Functional)                                           (None, 10)
#>
#>  model_70 (Functional)                                           (None, 20)
#>
#> ============================================================================================
#> Total params: 430
#> Trainable params: 430
#> Non-trainable params: 0
#> ____________________________________________________________________________________________
#> Model: "model_72"
#> ____________________________________________________________________________________________
#>  Layer (type)                                                    Output Shape
#> ============================================================================================
#>  prot.mod (InputLayer)                                           [(None, 142)]
#>
#>  model_63 (Functional)                                           (None, 50)
#>
#>  model_66 (Functional)                                           (None, 20)
#>
#>  model_69 (Functional)                                           (None, 10)
#>
#>  dense_80 (Dense)                                                (None, 5)
#>
#>  dense_79 (Dense)                                                (None, 1)
#>
#> ============================================================================================
#> Total params: 8,441
#> Trainable params: 8,441
#> Non-trainable params: 0
#> ____________________________________________________________________________________________
#> Model: "model_72"
#> ____________________________________________________________________________________________
#>  Layer (type)                                                    Output Shape
#> ============================================================================================
#>  prot.mod (InputLayer)                                           [(None, 142)]
#>
#>  model_63 (Functional)                                           (None, 50)
#>
#>  model_66 (Functional)                                           (None, 20)
#>
#>  model_69 (Functional)                                           (None, 10)
#>
#>  dense_80 (Dense)                                                (None, 5)
#>
#>  dense_79 (Dense)                                                (None, 1)
#>
#> ============================================================================================
#> Total params: 8,441
#> Trainable params: 61
#> Non-trainable params: 8,380
#> ______________________________________________________________________________________________________
```

```
#> Setting levels: control = 0, case = 1
```

```
#> Warning in roc.default(response = ytestlabels, predictor = yhat): Deprecated use a matrix as predict
#> be produced, please pass a numeric vector.
```

```
#> Setting direction: controls < cases
```



Concatenated Model

In this section we concatenate the two SAEs to fit, on the trainset, a DNN that integrates both data sources to predict estrogen receptor status. The DNN has a dense layer, with 20 nodes, which where the best amount of nodes found with **tfruns** earlier, in addition to the output layer.

The two models are concatenated below.

```
sae_output3 <- sae_input %>%
  encoder1() %>%
  encoder2() %>%
  encoder3() %>%
  layer_dense(units=20, activation = "relu")

sae_protab_output <- sae_protab_input %>%
  prot_encoder1() %>%
  prot_encoder2() %>%
  prot_encoder3() %>%
  layer_dense(units=20, activation = "relu")
```

```
concatenated <- layer_concatenate(list(sae_output3,sae_protab_output))

final_model_output <- concatenated %>%
  layer_dense(units=2,activation="softmax")

model <- keras_model(list(sae_input,sae_protab_input), final_model_output)
summary(model)
```

```
#> Model: "model_73"
#> ____________________________________________________________________________________________________
#>  Layer (type)                        Output Shape              Param #        Connected to
#> ====================================================================================================
#>  gene.mod (InputLayer)               [(None, 4454)]            0              []
#>
#>  prot.mod (InputLayer)               [(None, 142)]             0              []
#>
#>  model_51 (Functional)               (None, 1000)              4455000        ['gene.mod[0][0]
#>
#>  model_63 (Functional)               (None, 50)                7150           ['prot.mod[0][0]
#>
#>  model_54 (Functional)               (None, 100)               100100         ['model_51[2][0]
#>
#>  model_66 (Functional)               (None, 20)                1020           ['model_63[2][0]
#>
#>  model_57 (Functional)               (None, 50)                5050           ['model_54[2][0]
#>
#>  model_69 (Functional)               (None, 10)                210            ['model_66[2][0]
#>
#>  dense_81 (Dense)                    (None, 20)                1020           ['model_57[2][0]
#>
#>  dense_82 (Dense)                    (None, 20)                220            ['model_69[2][0]
#>
#>  concatenate_8 (Concatenate)         (None, 40)                0              ['dense_81[0][0]
#>                                                                                'dense_82[0][0]
#>
#>  dense_83 (Dense)                    (None, 2)                 82             ['concatenate_8
#>
#> ====================================================================================================
#> Total params: 4,569,852
#> Trainable params: 1,322
#> Non-trainable params: 4,568,530
#> ____________________________________________________________________________________________________
```

As can be seen from the parameters of the model above, only the weights for the dense layers will be fine-tuned during training of the concatenated model, in accordance with the procedure used earlier in the task.

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = "acc"
)

model %>% fit(
  x=list(as.matrix(xgene.train),as.matrix(xprot.train)),
  y=ytrain.bin,
  epochs = 90,
  batch_size = 64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

Performance Metrics

The model is evaluated on the test set below.

```
model %>%
  evaluate(x=list(as.matrix(xgene.test),as.matrix(xprot.test)),
  y=ytest.bin)
```

```
#>      loss       acc
#> 0.5425067 0.7037037
```

Predictions on the test set are calculated. The classifier is built on the assumption that predictions of probability smaller than 0.5 are negative receptor states, while probabilities larger than 0.5 are positive receptor states. The confusion matrix of the predictions is shown below.

```
yhat <- as.array(model %>% predict(list(as.matrix(xgene.test),as.matrix(xprot.test))) %>% k_argmax())
yhatclass <- as.factor(yhat)
confusionMatrix(yhatclass,as.factor(ytestlabels.comp))
```

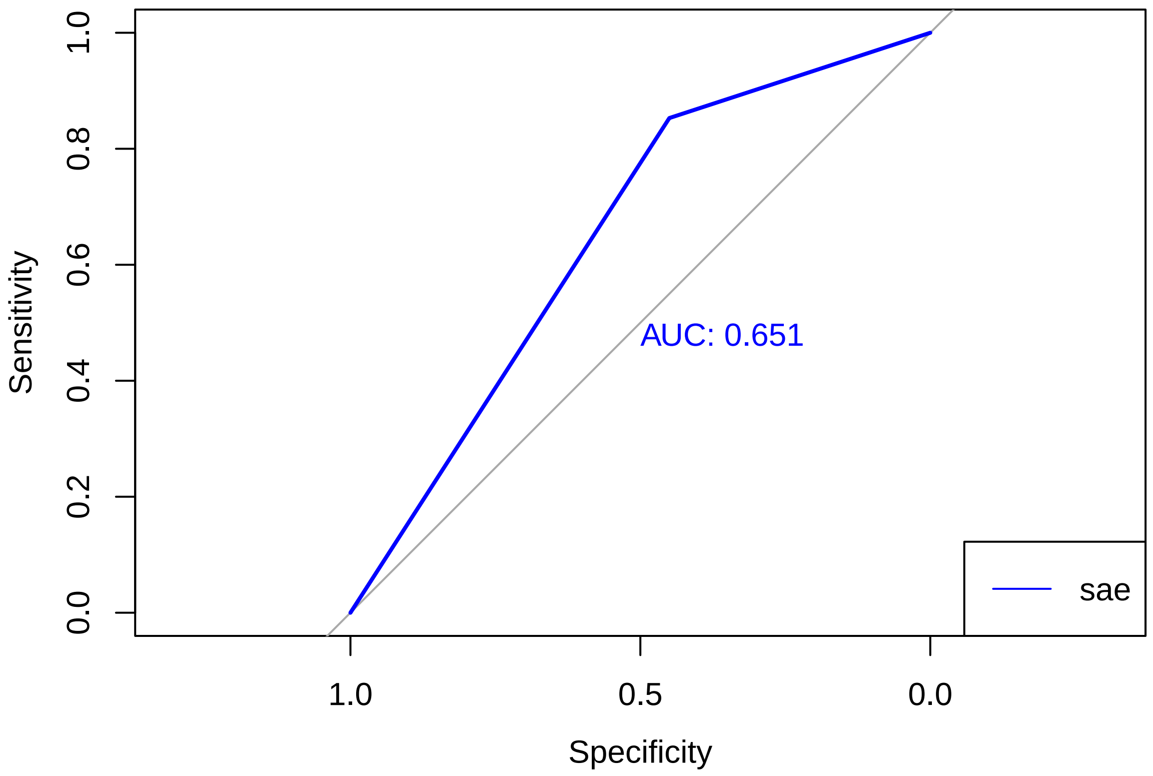
```
#> Confusion Matrix and Statistics
#> 
#>           Reference
#> Prediction  0  1
#>          0  9  5
#>          1 11 29
#> 
#>                Accuracy : 0.7037
#>                  95% CI : (0.5639, 0.8202)
#>     No Information Rate : 0.6296
#>     P-Value [Acc > NIR] : 0.1621
#> 
#>                   Kappa : 0.3229
#> 
#>  Mcnemar's Test P-Value : 0.2113
#> 
#>             Sensitivity : 0.4500
#>             Specificity : 0.8529
#>          Pos Pred Value : 0.6429
#>          Neg Pred Value : 0.7250
#>              Prevalence : 0.3704
#>          Detection Rate : 0.1667
#>    Detection Prevalence : 0.2593
#>       Balanced Accuracy : 0.6515
#> 
#>        'Positive' Class : 0
#> 
```

The ROC curve is shown below.

```r
roc_sae_test <- roc(response = ytestlabels.comp, predictor = as.numeric(yhat))
```

```
#> Setting levels: control = 0, case = 1
#> Setting direction: controls < cases
```

```r
plot(roc_sae_test, col = "blue", print.auc=TRUE)
legend("bottomright", legend = c("sae"), lty = c(1), col = c("blue"))
```



Discussion

There seems to be some different angles to address the first step of processing the data sets. The removal of the patients samples with missing values in the gene expression data set seems to be an important step to secure that all used patients has complete data to pass as input to the network. The question then arise if we only should include the patients samples that both exist in the gene expression data set and the protein abundance data set when implementing the stacked autoencoder, to then be able to use the same set for the concatenate version in the later stage. Since the first part asked specific to implement a SAE with the gene expression data we choose the alternative that was to include all complete patients in the gene expression data set, with no consideration of they existed in the protein abundance data set. Or motivation was that this should work as least as good as the alternative for the SAE, this method gave more patients left in the actual used training set.

One thing to remember when choosing the above described way is that we should make sure that the same genes that were selected for the training of the first SAE are also the selected in the preparation of the joint data set for the concatenate SAE. Otherwise we risk training the first SAE on one set of genes from each patient and then using it on a potential other set of genes.

In the training of the SAE one difficulty is to avoid both underfitting and overfitting on the training data. With the aim to run each training for a suitable amount of epochs each training were monitored so that it could not keep training the network if the performance on the validation set did not increase during a fixed number of epochs. The intention is that this will lead to a network with better performance on a test set.