# Task1

Statistical Learning with Deep Artificial Neural Networks

Liv Breivik, Hannes Johansson, Alexander J Ohrt

11. 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 Datasets

```
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 dataset are shown below.

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

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

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

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

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

The dimensions of the gene expression dataset are shown below.

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

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

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

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

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

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

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

```
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 its own sampling code. After the rows with missing data are removed, the gene expression data contains 247 unique patients.

Next we find which patients have 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

```
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 dataset. Column 9 contains the response we want to predict, which has the following unique values

```
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 preprocessed later. Below the code used to find patients that have data available is given.

```
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 have recorded the response. Notice that this the data that will be used in the first part of the task.

```
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 datasets, before removing the rows with missing values from the gene expression data.

```
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.

```
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` dataset. Thus, this contains the unique list of patients that have all necessary data in all three datasets. Notice that these patients will be used to define the complete dataset, 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` dataset contains. Next, we remove all the individuals that don't have this information.

```
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 not negative or 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.
```

```r
# 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 last preprocess, this dataset now contains the patients that have the complete gene expression data, as well as a well-defined BRCA receptor status. In the later parts

### 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 other set in section 4 onwards, to make sure that the same set of genes that the network was trained with are the ones selected for that set as well.

```r
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

```r
set.seed(111)
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_78"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_64 (InputLayer)              [(None, 4454)]                  0
#>
#>  dense_84 (Dense)                   (None, 1000)                    4455000
#>
#> ================================================================================
#> 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_79"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_65 (InputLayer)              [(None, 1000)]                  0
#>
#>  dense_85 (Dense)                   (None, 4454)                    4458454
#>
#> ================================================================================
#> 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_80"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_66 (InputLayer)              [(None, 4454)]                  0
#>
#>  model_78 (Functional)              (None, 1000)                    4455000
#>
#>  model_79 (Functional)              (None, 4454)                    4458454
#>
#> ================================================================================
#> 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
)
```

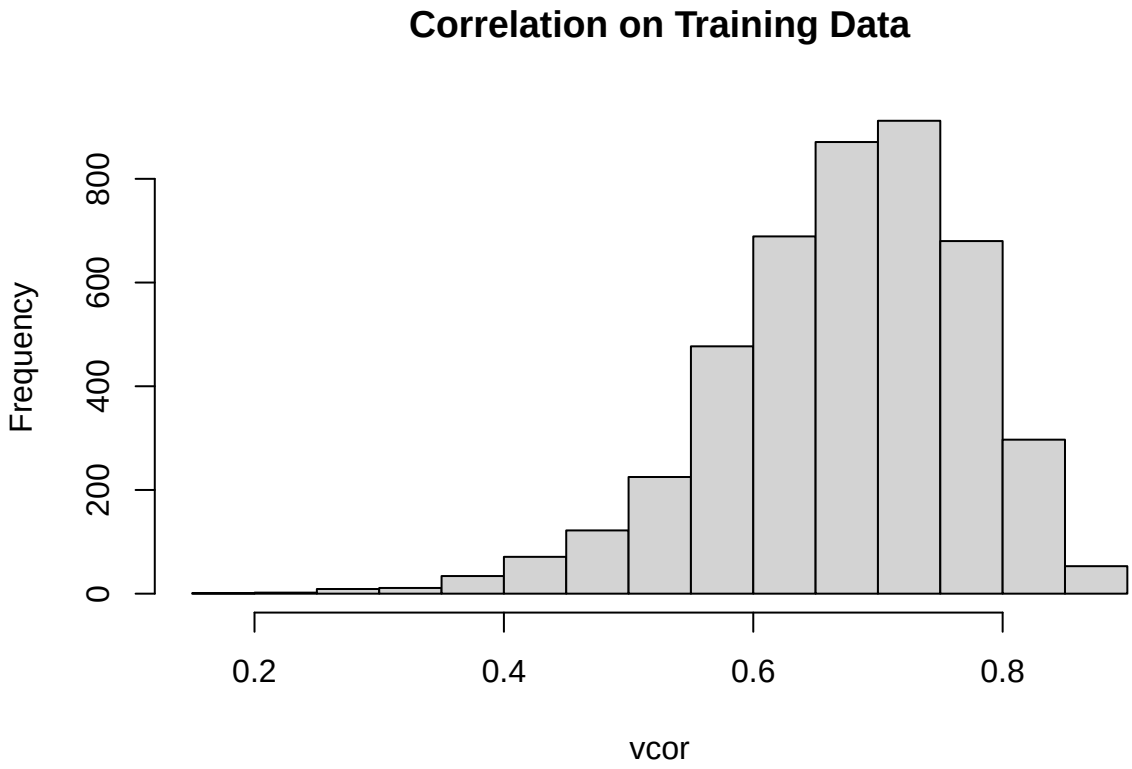
We make predictions on the training data.

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

```
# 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)
```

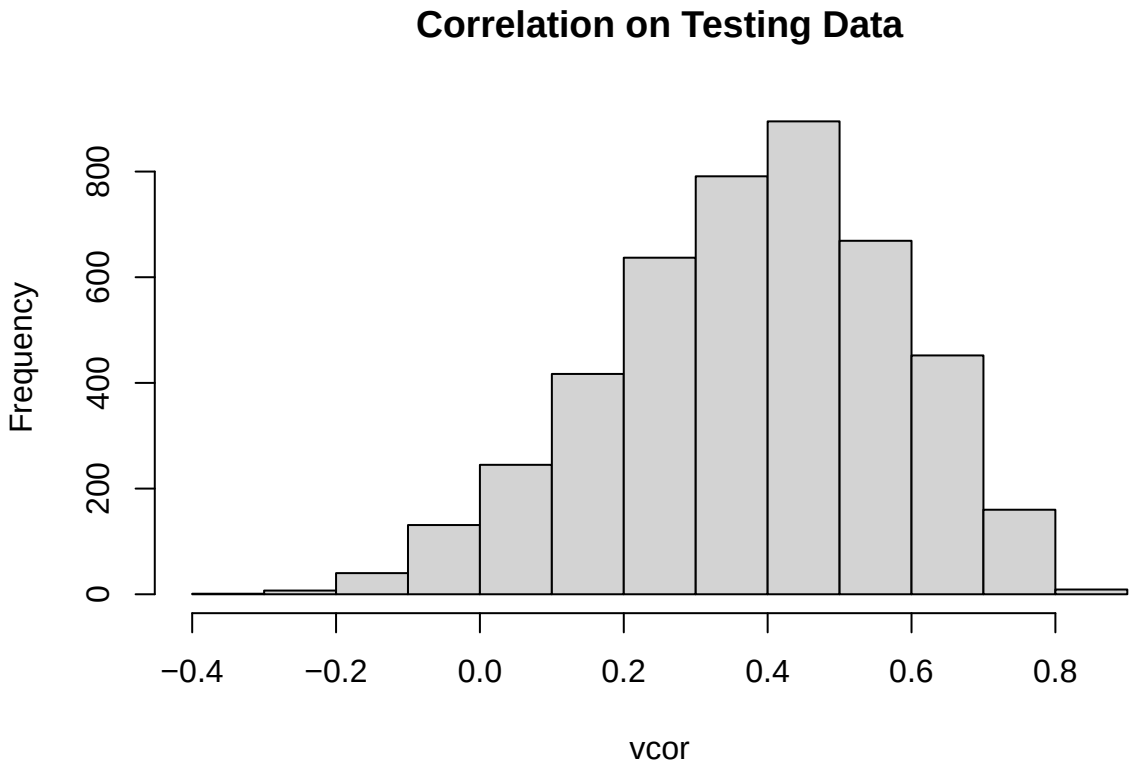
Some (weak) evidence of the quality of the coding obtained follows. We plot the correlation between the predictions from the autoencoder and the correct data, both on the train and test sets. The results are shown below.

```
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 menitioning 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.

```
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_81"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_67 (InputLayer)              [(None, 1000)]                  0
#>
#>  dense_86 (Dense)                   (None, 100)                     100100
#>
#> ================================================================================
#> 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_82"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_68 (InputLayer)               [(None, 100)]                     0
#>
#>  dense_87 (Dense)                    (None, 1000)                      101000
#>
#> =================================================================================
#> 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_83"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_69 (InputLayer)               [(None, 1000)]                    0
#>
#>  model_81 (Functional)               (None, 100)                       100100
#>
#>  model_82 (Functional)               (None, 1000)                      101000
#>
#> =================================================================================
#> 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(
```

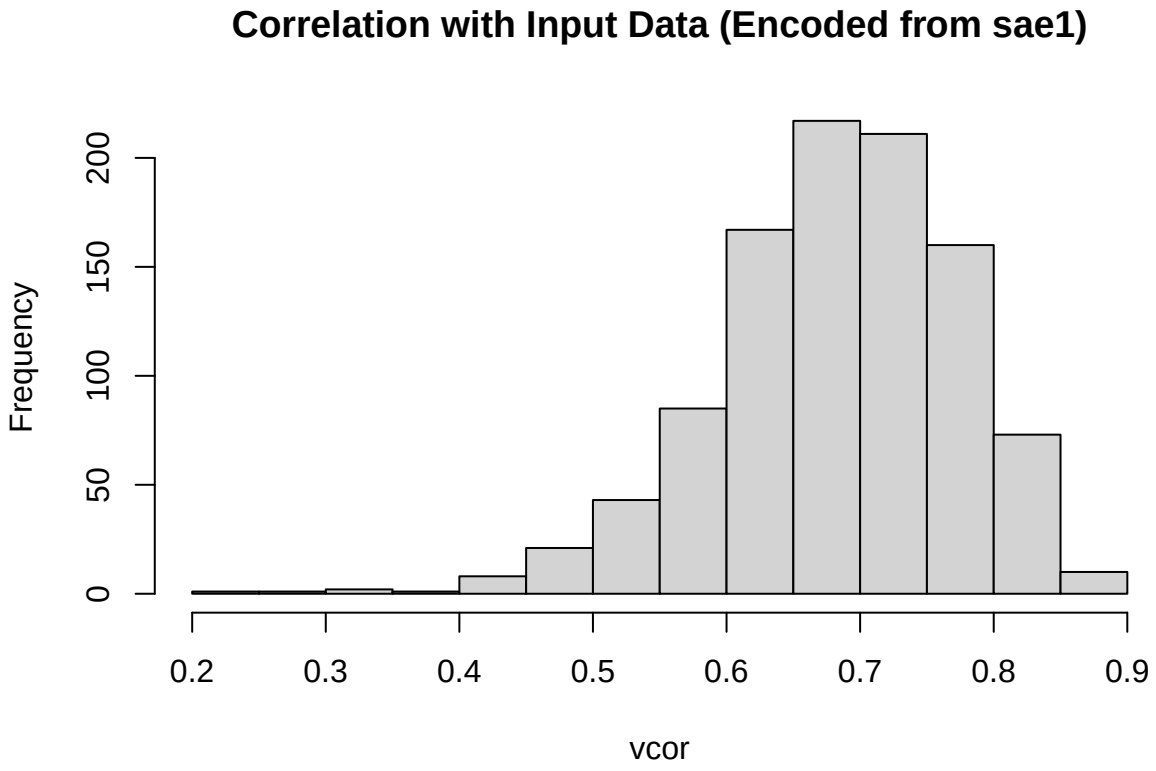
```
  x = encoded_expression1,
  y = encoded_expression1,
  epochs = 70,
  batch_size = 64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

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)

```
# 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_84"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_70 (InputLayer)              [(None, 100)]                   0
#>
#>  dense_88 (Dense)                   (None, 50)                      5050
#>
#> ================================================================================
#> Total params: 5,050
#> Trainable params: 5,050
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

```
# 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_85"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_71 (InputLayer)              [(None, 50)]                    0
#>
#>  dense_89 (Dense)                   (None, 100)                     5100
#>
#> ================================================================================
#> Total params: 5,100
#> Trainable params: 5,100
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

```
# 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_86"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
```

```
#>  input_72 (InputLayer)              [(None, 100)]                   0
#>
#>  model_84 (Functional)              (None, 50)                      5050
#>
#>  model_85 (Functional)              (None, 100)                     5100
#>
#> ================================================================================
#> Total params: 10,150
#> Trainable params: 10,150
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

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

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

```
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.

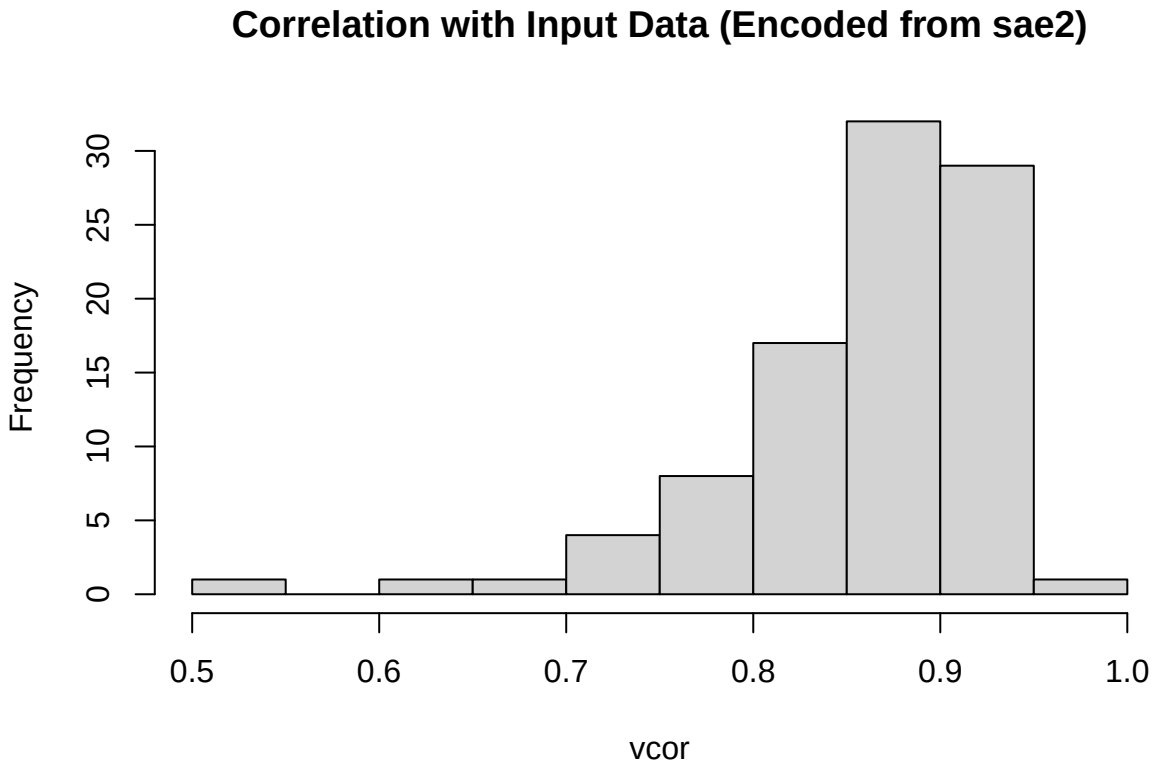
```
encoded_expression3 <- encoder3 %>% predict(encoded_expression2)
decoded_expression3 <- decoder3 %>% predict(encoded_expression3)

# 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).

```
vcor <- diag(cor(x.hat,encoded_expression2))
```

```
#> Warning in cor(x.hat, encoded_expression2): the standard deviation is zero
```

```
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 are stacked 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_87"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  gene.mod (InputLayer)              [(None, 4454)]                  0
#>
#>  model_78 (Functional)              (None, 1000)                    4455000
#>
#>  model_81 (Functional)              (None, 100)                     100100
#>
#>  model_84 (Functional)              (None, 50)                      5050
#>
```

```
#> ================================================================================
#> 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) 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_88"
#> --------------------------------------------------------------------------------
#>  Layer (type)                        Output Shape                    Param #
#> ================================================================================
#>  gene.mod (InputLayer)               [(None, 4454)]                  0
#>
#>  model_78 (Functional)               (None, 1000)                    4455000
#>
#>  model_81 (Functional)               (None, 100)                     100100
#>
#>  model_84 (Functional)               (None, 50)                      5050
#>
#>  dense_91 (Dense)                    (None, 10)                      510
#>
#>  dense_90 (Dense)                    (None, 1)                       11
#>
#> ================================================================================
#> 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).
summary(sae)
```

```
#> Model: "model_88"
#> --------------------------------------------------------------------------------
#>  Layer (type)                        Output Shape                    Param #
#> ================================================================================
```

```
#>  gene.mod (InputLayer)              [(None, 4454)]                  0
#>
#>  model_78 (Functional)              (None, 1000)                    4455000
#>
#>  model_81 (Functional)              (None, 100)                     100100
#>
#>  model_84 (Functional)              (None, 50)                      5050
#>
#>  dense_91 (Dense)                   (None, 10)                      510
#>
#>  dense_90 (Dense)                   (None, 1)                       11
#>
#> ================================================================================
#> 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 = 80,
  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
#> 0.2844684 0.9000000
```

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 14  2
#>          1  5 49
#>
#>                Accuracy : 0.9
#>                  95% CI : (0.8048, 0.9588)
#>     No Information Rate : 0.7286
#>     P-Value [Acc > NIR] : 0.0003901
#>
#>                   Kappa : 0.734
#>
#>  Mcnemar's Test P-Value : 0.4496918
#>
#>             Sensitivity : 0.7368
#>             Specificity : 0.9608
#>          Pos Pred Value : 0.8750
#>          Neg Pred Value : 0.9074
#>              Prevalence : 0.2714
#>          Detection Rate : 0.2000
#>    Detection Prevalence : 0.2286
#>       Balanced Accuracy : 0.8488
#>
#>        'Positive' Class : 0
#>
```

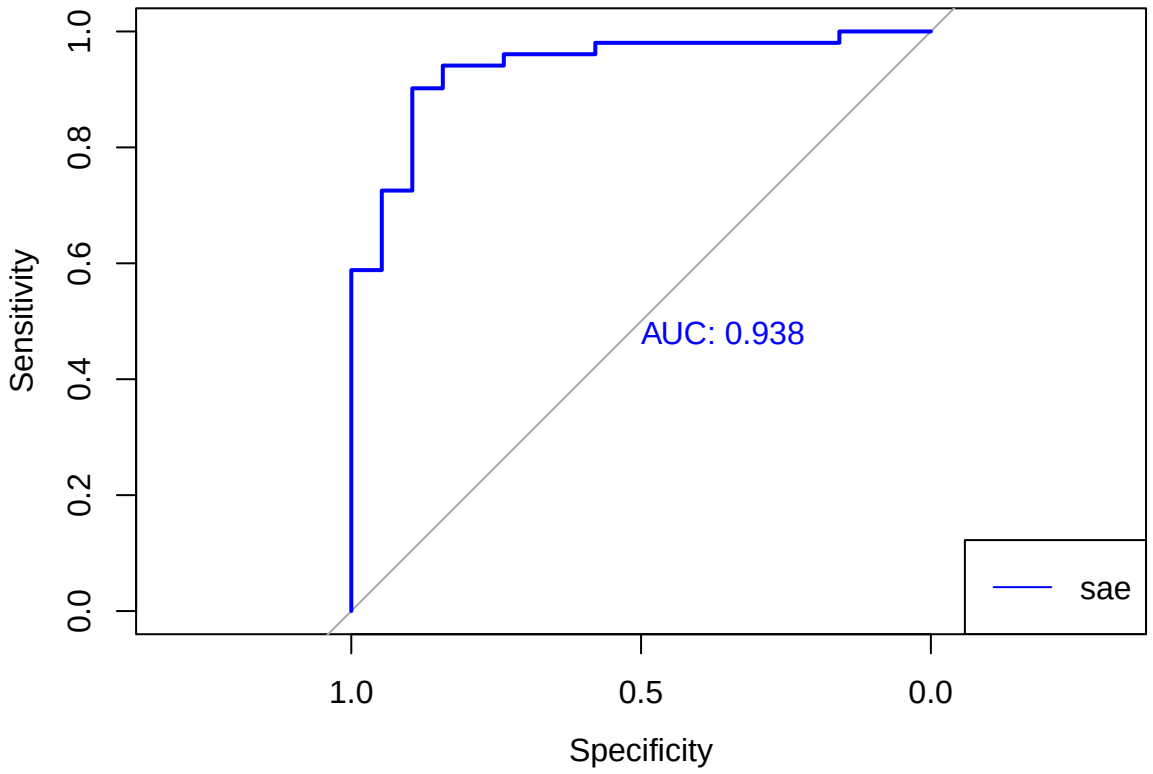
The ROC curve is shown below.

```
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. COMMENTS / INTERPRETATIONS!

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. Anyhow, 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_89"
#> ________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> ================================================================================
#>  gene.mod (InputLayer)               [(None, 4454)]                    0
#>
```

```
#>  model_78 (Functional)              (None, 1000)                     4455000
#>
#>  model_81 (Functional)              (None, 100)                      100100
#>
#>  model_84 (Functional)              (None, 50)                       5050
#>
#>  dense_93 (Dense)                   (None, 20)                       1020
#>
#>  dense_92 (Dense)                   (None, 1)                        21
#>
#> ================================================================================
#> Total params: 4,561,191
#> Trainable params: 1,041
#> Non-trainable params: 4,560,150
#> ________________________________________________________________________________
```

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

```
#> Model: "model_89"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                     Param #
#> ================================================================================
#>  gene.mod (InputLayer)              [(None, 4454)]                   0
#>
#>  model_78 (Functional)              (None, 1000)                     4455000
#>
#>  model_81 (Functional)              (None, 100)                      100100
#>
#>  model_84 (Functional)              (None, 50)                       5050
#>
#>  dense_93 (Dense)                   (None, 20)                       1020
#>
#>  dense_92 (Dense)                   (None, 1)                        21
#>
#> ================================================================================
#> Total params: 4,561,191
#> Trainable params: 1,041
#> Non-trainable params: 4,560,150
#> ________________________________________________________________________________
```

```
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
#> 0.2434354 0.9285714
```

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

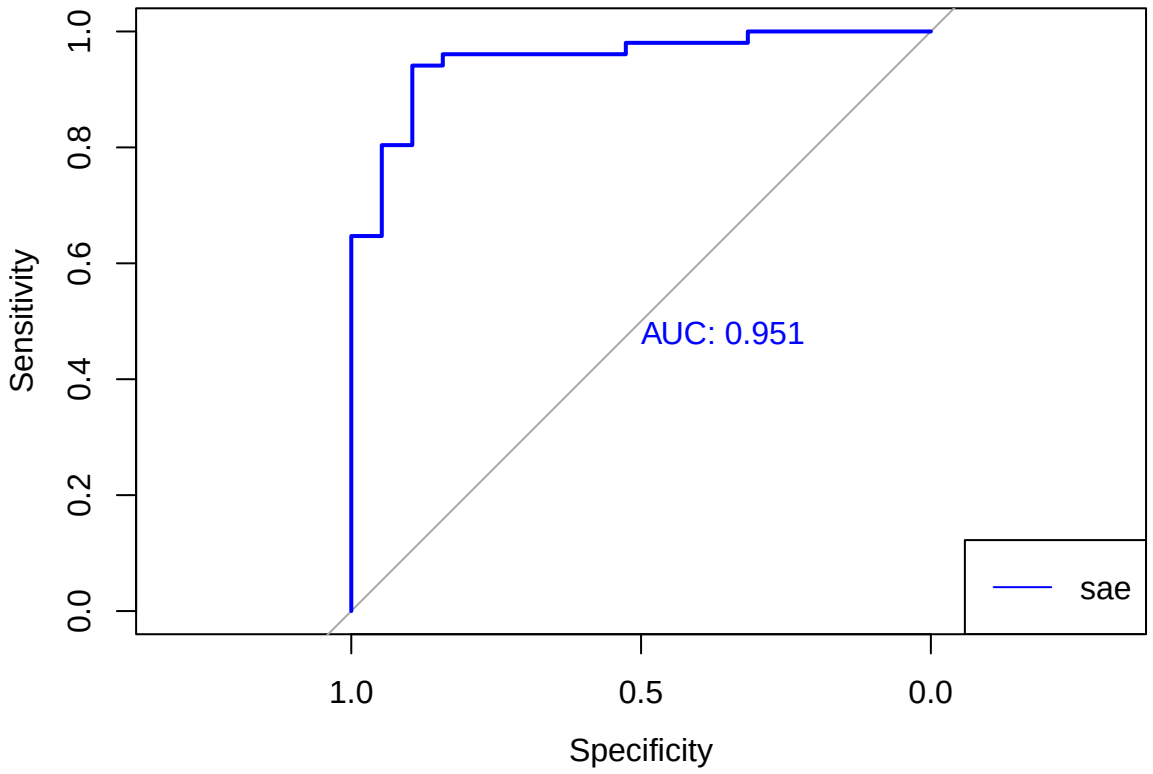
```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  0  1
#>          0 16  2
#>          1  3 49
#>
#>                Accuracy : 0.9286
#>                  95% CI : (0.8411, 0.9764)
#>     No Information Rate : 0.7286
#>     P-Value [Acc > NIR] : 2.543e-05
#>
#>                   Kappa : 0.8164
#>
#>  Mcnemar's Test P-Value : 1
#>
#>             Sensitivity : 0.8421
#>             Specificity : 0.9608
#>          Pos Pred Value : 0.8889
#>          Neg Pred Value : 0.9423
#>              Prevalence : 0.2714
#>          Detection Rate : 0.2286
#>    Detection Prevalence : 0.2571
#>       Balanced Accuracy : 0.9014
#>
#>        '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.

DENNE ER JO DÅRLIGERE. HVORDAN ORDNE OPP IT STOKASTISITET I TFRUNS + HER? HVILKEN VARIANT SOM ER BEST (5, 10 ELLER 20 NODER) VIRKER VELDIG TILFELDIG (OG AVHENGER OGSÅ STERKT AV HVOR MANGE VEKTER SOM FRYSES). I TILLEGG VIRKER DET IKKE SOM AT metric_val_acc ER ET GODT MÅL PÅ HVILKEN MODELL SOM GIR BEST RESULTATER, DA AUC OG ROC I FLERE TILFELLER VISER NOE ANNET.

??? Liv her, AUC nærmere 1 betyr jo at den er bedre? ... må huske å seede kanskje btw

## Complete Data

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

ATT LÄGGA IN VID DEN MOTSVARANDE KODEN: As explained in section 1, the same set of gene features is used in this dataset, 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 # Change this definition after deciding which data set to use!
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 not negative or positive.
# In this case these values are either "Indeterminate" or "Not Performed".
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.
```

```r
# 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.

set.seed(111)
training.fraction <- 0.70 # 70 % of data will be used for training.
training <- sample(1:nrow(mtot),nrow(mtot)*training.fraction)

xtrain <- mtot[training,-c(1,2)]
xtest <- mtot[-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
```

```r
xprot_train <- xprot[training,-c(1,2)]
xgene_train <- xgene[training,-c(1,2)]
xprot_test <- xprot[-training,-c(1,2)]
xgene_test <- xgene[-training,-c(1,2)]

ytrain_bin <- to_categorical(as.array(ylabels), 2)
```

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.

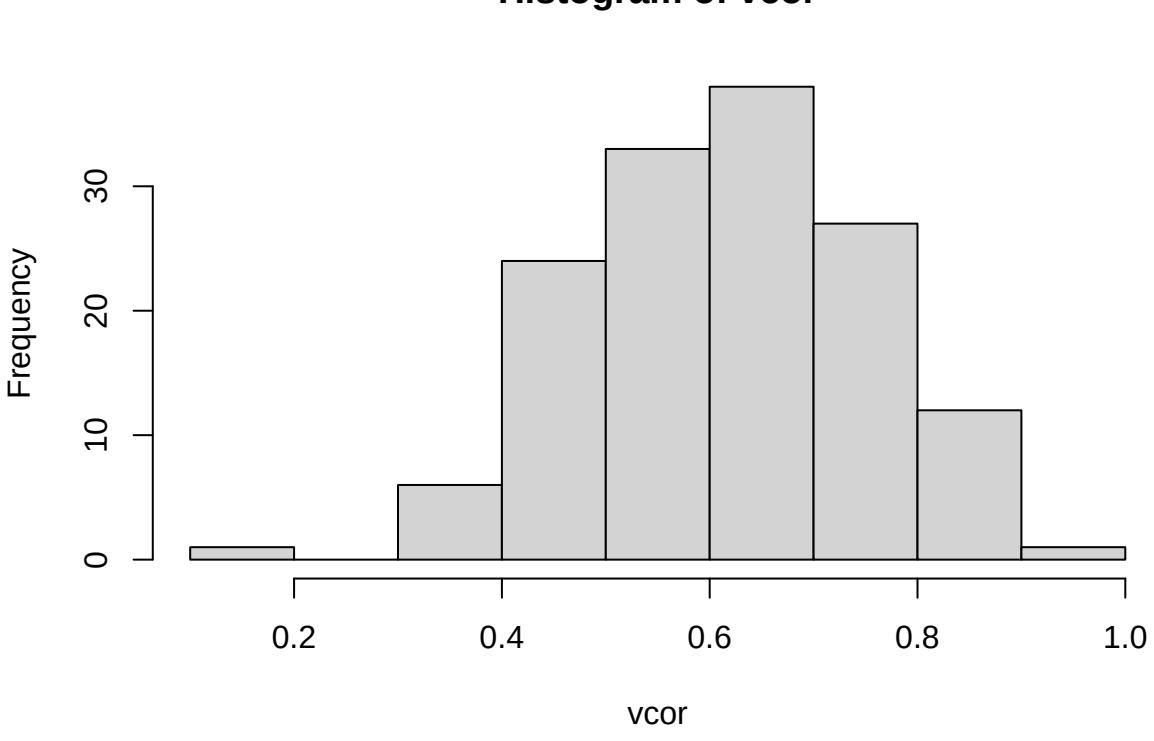
### Importing the SAE from the class example.

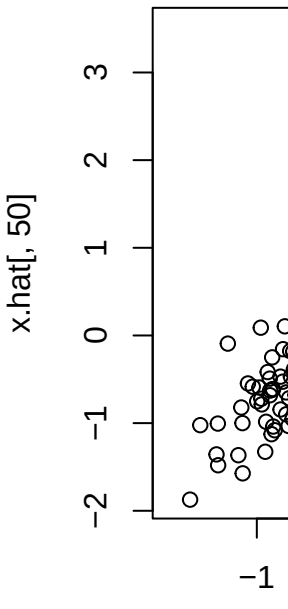
How I understood it: in the file aes_practice_3.R there is a SAE model that we can use for the breast stuff. We need to import this model and concatenate it with the one we created for gene expression.

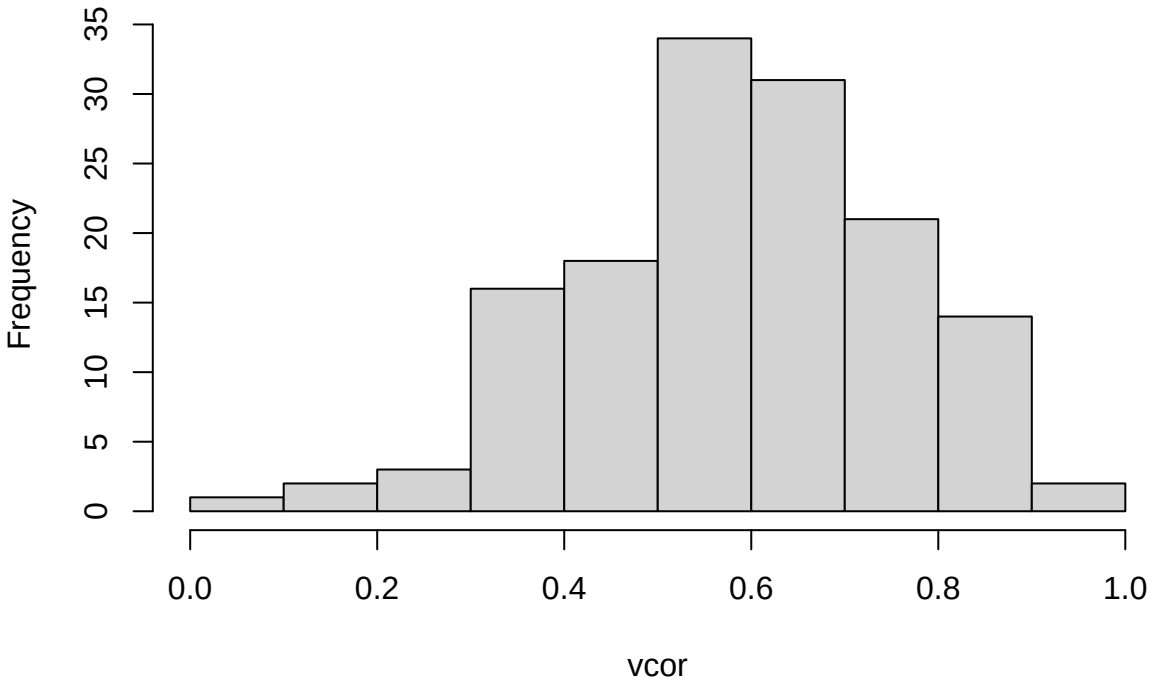
```r
# Model given in class.
source("aes_practice_3.R") # source: Parses the code and evaluates it in this environment.
```

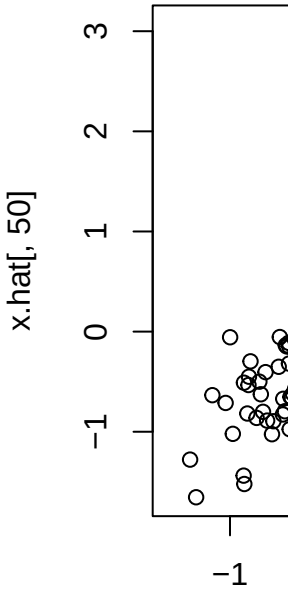
```
#> Rows: 847 Columns: 19
#> -- Column specification --------------------------------------------------------
#> Delimiter: "\t"
#> chr (19): Sample, Histology, PAM50Call, ajcc_cancer_metastasis_stage_code, ajcc_...
#>
#> i Use `spec()` to retrieve the full column specification for this data.
```

```
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
#> 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.
#> Model: "sequential_3"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  dense_97 (Dense)                    (None, 50)                        7150
#>
#>  dense_96 (Dense)                    (None, 20)                        1020
#>
#>  dense_95 (Dense)                    (None, 50)                        1050
#>
#>  dense_94 (Dense)                    (None, 142)                       7242
#>
#> =================================================================================
#> Total params: 16,462
#> Trainable params: 16,462
#> Non-trainable params: 0
#> _________________________________________________________________________________
```





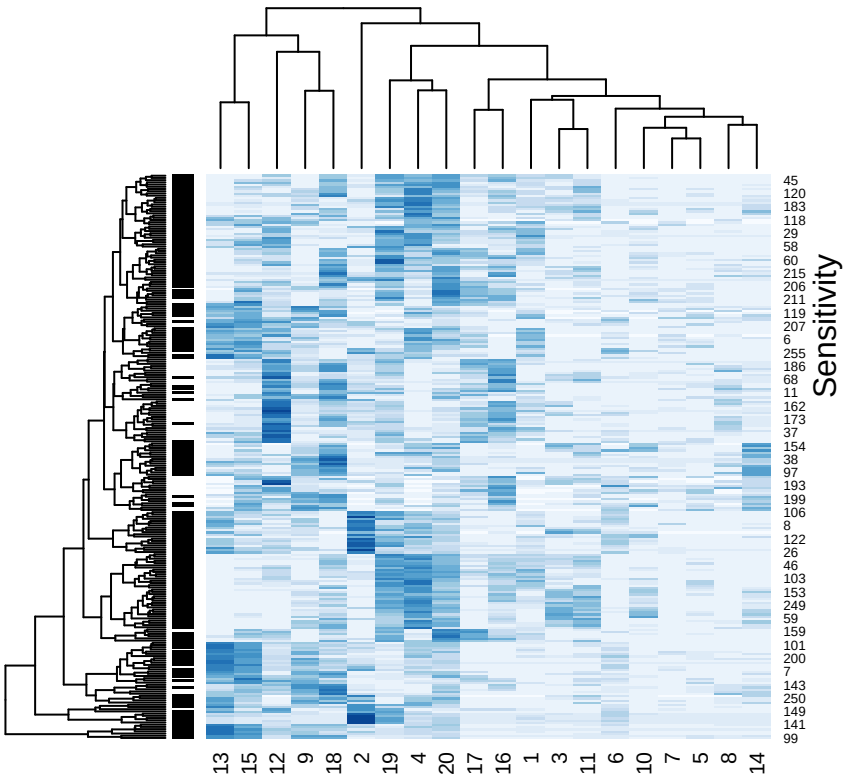

```
#> Model: "model_90"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_73 (InputLayer)              [(None, 142)]                   0
#>
#>  dense_99 (Dense)                   (None, 50)                      7150
#>
#>  dense_98 (Dense)                   (None, 20)                      1020
#>
#> ================================================================================
#> Total params: 8,170
#> Trainable params: 8,170
#> Non-trainable params: 0
#> ________________________________________________________________________________
#> Model: "model_91"
#> ________________________________________________________________________________
#>  Layer (type)                       Output Shape                    Param #
#> ================================================================================
#>  input_74 (InputLayer)              [(None, 20)]                    0
#>
#>  dense_101 (Dense)                  (None, 50)                      1050
#>
#>  dense_100 (Dense)                  (None, 142)                     7242
#>
#> ================================================================================
#> Total params: 8,292
#> Trainable params: 8,292
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

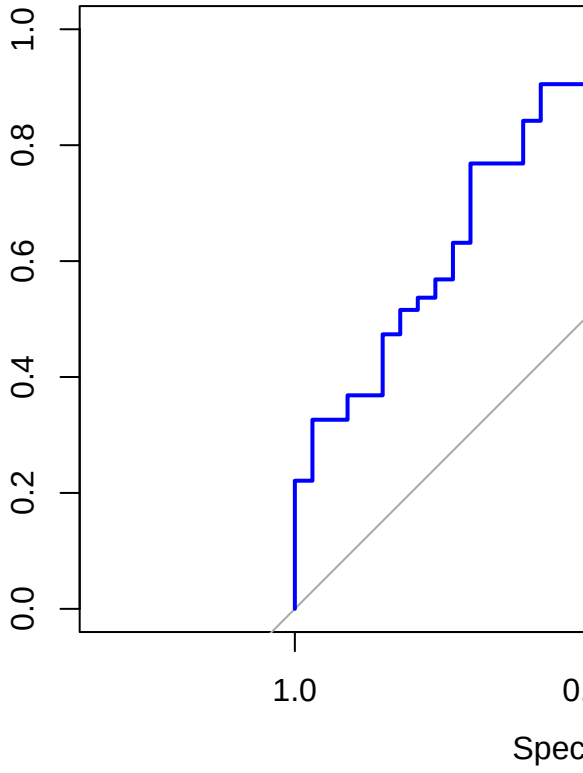
```
#> Model: "model_92"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_75 (InputLayer)               [(None, 142)]                     0
#>
#>  model_90 (Functional)               (None, 20)                        8170
#>
#>  model_91 (Functional)               (None, 142)                       8292
#>
#> =================================================================================
#> Total params: 16,462
#> Trainable params: 16,462
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_93"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_76 (InputLayer)               [(None, 142)]                     0
#>
#>  dense_102 (Dense)                   (None, 50)                        7150
#>
#> =================================================================================
#> Total params: 7,150
#> Trainable params: 7,150
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_94"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_77 (InputLayer)               [(None, 50)]                      0
#>
#>  dense_103 (Dense)                   (None, 142)                       7242
#>
#> =================================================================================
#> Total params: 7,242
#> Trainable params: 7,242
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_95"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_78 (InputLayer)               [(None, 142)]                     0
#>
#>  model_93 (Functional)               (None, 50)                        7150
#>
#>  model_94 (Functional)               (None, 142)                       7242
#>
#> =================================================================================
#> Total params: 14,392
```

```
#> Trainable params: 14,392
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_96"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_79 (InputLayer)               [(None, 50)]                      0
#>
#>  dense_104 (Dense)                   (None, 20)                        1020
#>
#> =================================================================================
#> Total params: 1,020
#> Trainable params: 1,020
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_97"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_80 (InputLayer)               [(None, 20)]                      0
#>
#>  dense_105 (Dense)                   (None, 50)                        1050
#>
#> =================================================================================
#> Total params: 1,050
#> Trainable params: 1,050
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_98"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_81 (InputLayer)               [(None, 50)]                      0
#>
#>  model_96 (Functional)               (None, 20)                        1020
#>
#>  model_97 (Functional)               (None, 50)                        1050
#>
#> =================================================================================
#> Total params: 2,070
#> Trainable params: 2,070
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_99"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                      Param #
#> =================================================================================
#>  input_82 (InputLayer)               [(None, 20)]                      0
#>
#>  dense_106 (Dense)                   (None, 10)                        210
#>
#> =================================================================================
#> Total params: 210
```

```
#> Trainable params: 210
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_100"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                       Param #
#> =================================================================================
#>  input_83 (InputLayer)               [(None, 10)]                       0
#>
#>  dense_107 (Dense)                   (None, 20)                         220
#>
#> =================================================================================
#> Total params: 220
#> Trainable params: 220
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_101"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                       Param #
#> =================================================================================
#>  input_84 (InputLayer)               [(None, 20)]                       0
#>
#>  model_99 (Functional)               (None, 10)                         210
#>
#>  model_100 (Functional)              (None, 20)                         220
#>
#> =================================================================================
#> Total params: 430
#> Trainable params: 430
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_102"
#> _________________________________________________________________________________
#>  Layer (type)                        Output Shape                       Param #
#> =================================================================================
#>  prot.mod (InputLayer)               [(None, 142)]                      0
#>
#>  model_93 (Functional)               (None, 50)                         7150
#>
#>  model_96 (Functional)               (None, 20)                         1020
#>
#>  model_99 (Functional)               (None, 10)                         210
#>
#>  dense_109 (Dense)                   (None, 5)                          55
#>
#>  dense_108 (Dense)                   (None, 1)                          6
#>
#> =================================================================================
#> Total params: 8,441
#> Trainable params: 8,441
#> Non-trainable params: 0
#> _________________________________________________________________________________
#> Model: "model_102"
#> _________________________________________________________________________________
```

```
#>  Layer (type)                        Output Shape                      Param #
#> ================================================================================
#>  prot.mod (InputLayer)               [(None, 142)]                     0
#>
#>  model_93 (Functional)               (None, 50)                        7150
#>
#>  model_96 (Functional)               (None, 20)                        1020
#>
#>  model_99 (Functional)               (None, 10)                        210
#>
#>  dense_109 (Dense)                   (None, 5)                         55
#>
#>  dense_108 (Dense)                   (None, 1)                         6
#>
#> ================================================================================
#> 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 predictor. Unexpected results may 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.

```
concatenated<-layer_concatenate(list(sae_output2,sae_protab_output))
model_output<-concatenated %>%
  layer_dense(20,"relu") %>%
  layer_dense(units=1,activation="softmax")

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

```
#> Model: "model_103"
#> ____________________________________________________________________________________
#>  Layer (type)                Output Shape              Param #   Connected to
#> ====================================================================================
#>  gene.mod (InputLayer)       [(None, 4454)]            0         []
#>
#>  prot.mod (InputLayer)       [(None, 142)]             0         []
#>
#>  model_78 (Functional)       (None, 1000)              4455000   ['gene.mod[0][0]']
#>
#>  model_93 (Functional)       (None, 50)                7150      ['prot.mod[0][0]']
#>
#>  model_81 (Functional)       (None, 100)               100100    ['model_78[1][0]']
#>
#>  model_96 (Functional)       (None, 20)                1020      ['model_93[1][0]']
#>
#>  model_84 (Functional)       (None, 50)                5050      ['model_81[1][0]']
#>
#>  model_99 (Functional)       (None, 10)                210       ['model_96[1][0]']
#>
#>  dense_93 (Dense)            (None, 20)                1020      ['model_84[1][0]']
#>
#>  dense_109 (Dense)           (None, 5)                 55        ['model_99[1][0]']
#>
#>  dense_92 (Dense)            (None, 1)                 21        ['dense_93[0][0]']
#>
#>  dense_108 (Dense)           (None, 1)                 6         ['dense_109[0][0]']
#>
#>  concatenate_3 (Concatenat  (None, 2)                  0         ['dense_92[0][0]',
#>  e)                                                               'dense_108[0][0]']
#>
#>  dense_111 (Dense)           (None, 20)                60        ['concatenate_3[0][0]']
#>
#>  dense_110 (Dense)           (None, 1)                 21        ['dense_111[0][0]']
#>
#> ====================================================================================
#> Total params: 4,569,713
#> Trainable params: 1,183
#> Non-trainable params: 4,568,530
#> ____________________________________________________________________________________
```

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

# training
#Den her funker ikke. Jeg tror det er fordi xgene_train har 17000 ish kolonner. Den burde egentlig ha t
model %>% fit(
  list(gene.mod=xgene_train, prot.mod=xprot_train), as.array(ylabels),
  epochs = 75, batch_size = 64, validation_split = 0.2
)
```

KODEN NEDENFOR FUNGERER IKKE AKKURAT NÅ, USIKKER PÅ HVORFOR DEN FAILER.

Liv her: jeg tror den failer pga denne SAE skal kun brukes på protein abundance greia. Egt ville det beste vært om vi kunne eksportert modellen fra scriptet til læreren, siden det er nøyaktig det vi skal bruke. Hvis ikke burde vi ikke definere xtrain som noe nytt.

ˆGjort ovenfor. Koden nedenfor kan slettes, men jeg har ikke lyst til å slette andre sitt arbeid. (Vi sitter nå med modellene sae og sae_protab)

```
## ------------------------------------------------------------------------------------
input_enc1<-layer_input(shape = c(142))
output_enc1<-input_enc1 %>%
  layer_dense(units=50,activation="relu")
encoder1 = keras_model(input_enc1, output_enc1)
summary(encoder1)


## ------------------------------------------------------------------------------------
input_dec1 = layer_input(shape = c(50))
output_dec1<-input_dec1 %>%
  layer_dense(units=142,activation="linear")

decoder1 = keras_model(input_dec1, output_dec1)

summary(decoder1)


## ------------------------------------------------------------------------------------
aen_input1 = layer_input(shape = c(142))
aen_output1 = aen_input1 %>%
  encoder1() %>%
  decoder1()

sae1 = keras_model(aen_input1, aen_output1)
summary(sae1)


## ------------------------------------------------------------------------------------
sae1 %>% compile(
  loss = "mse",
  optimizer = "rmsprop",
  metrics = c('accuracy')
```

```
)


## ------------------------------------------------------------------------------------
sae1 %>% fit(
  x=as.matrix(xtrain),
  y=ylabels,
  epochs = 100,
  batch_size=64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
  )
#y=as.matrix(xtrain)


## ------------------------------------------------------------------------------------
#Generating with Autoencoder
encoded_expression1 <- encoder1 %>% predict(as.matrix(xtrain))


## ------------------------------------------------------------------------------------
input_enc2<-layer_input(shape = c(50))
output_enc2<-input_enc2 %>%
  layer_dense(units=20,activation="relu")
encoder2 = keras_model(input_enc2, output_enc2)
summary(encoder2)


## ------------------------------------------------------------------------------------
input_dec2 = layer_input(shape = c(20))
output_dec2<-input_dec2 %>%
  layer_dense(units=50,activation="linear")

decoder2 = keras_model(input_dec2, output_dec2)

summary(decoder2)


## ------------------------------------------------------------------------------------
aen_input2 = layer_input(shape = c(50))
aen_output2 = aen_input2 %>%
  encoder2() %>%
  decoder2()

sae2 = keras_model(aen_input2, aen_output2)
summary(sae2)


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

```
## ------------------------------------------------------------------------------------
sae2 %>% fit(
  x=as.matrix(encoded_expression1),
  y=as.matrix(encoded_expression1),
  epochs = 300,
  batch_size=64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
  )


## ------------------------------------------------------------------------------------
#Generating with Autoencoder
encoded_expression2 <- encoder2 %>% predict(as.matrix(encoded_expression1))


## ------------------------------------------------------------------------------------
input_enc3<-layer_input(shape = c(20))
output_enc3<-input_enc3 %>%
  layer_dense(units=10,activation="relu")
encoder3 = keras_model(input_enc3, output_enc3)
summary(encoder3)


## ------------------------------------------------------------------------------------
input_dec3 = layer_input(shape = c(10))
output_dec3<-input_dec3 %>%
  layer_dense(units=20,activation="linear")

decoder3 = keras_model(input_dec3, output_dec3)

summary(decoder3)


## ------------------------------------------------------------------------------------
aen_input3 = layer_input(shape = c(20))
aen_output3 = aen_input3 %>%
  encoder3() %>%
  decoder3()

sae3 = keras_model(aen_input3, aen_output3)
summary(sae3)


## ------------------------------------------------------------------------------------
sae3 %>% compile(
  optimizer = "rmsprop",
  loss = "mse"
)


## ------------------------------------------------------------------------------------
```

```
sae3 %>% fit(
  x=as.matrix(encoded_expression2),
  y=as.matrix(encoded_expression2),
  epochs = 300,
  batch_size=64,
  validation_split = 0.2,
  callbacks = callbacks_parameters
  )


## ------------------------------------------------------------------------------------
#Generating with Autoencoder
encoded_expression3 <- encoder3 %>% predict(as.matrix(encoded_expression2))

sae_input.prot = layer_input(shape = c(142), name = "prot.mod")
sae_output.prot = sae_input %>%
  encoder1() %>%
  encoder2()  %>%
  encoder3() %>%
  layer_dense(20,activation = "relu")%>%
  layer_dense(1,activation = "sigmoid")
```

The two models are concatenated below. CHECK THAT THIS IS DONE CORRECTLY LATER! SKAL DENSE LAYERS LEGGES TIL I DEN KONKATENERTE MODELLEN ELLER I HVER AV DE TO MODELLENE FØR DE KONKATINERES (SLIK DET ER GJORT NÅ)?

```
concatenated<-layer_concatenate(list(sae_output2,sae_output.prot))
model_output<-concatenated %>%
  layer_dense(units=2,activation = "softmax")

model<-keras_model(list(sae_input,sae_output.prot), model_output)
summary(model)
```

The model is compiled and trained below.

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


# training

model %>% fit( # Usikker på om det er rett at begge skal ha den samme dataen!?
  # Mulig dette blir helt feil dimensjoner!
  list(gene.mod=xtrain,prot.mod=xtrain), ylabels,
  epochs = 60, batch_size = 16, validation_split = 0.2,
  callbacks = callbacks_parameters
)
```

## Performance Metrics

The model is evaluated on the test set below.

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

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

The ROC curve is shown below.

```
roc_sae_test <- roc(response = ytestlabels, predictor = as.numeric(yhat))
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 datasets. The removal of the patients samples with missing values in the gene expression dataset 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 dataset and the protein abundance dataset 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 dataset, with no consideration of they existed in the protein abundance dataset. 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 dataset 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.

WE CURRENTLY CANNOT MAKE ANY DISCUSSION OR COMPARISON ABOUT THE PERFORMANCE CONCATENATED MODEL.