STA4026S Assignment 2 - Collider Data Classification

Shriyaa Sooklal SKLSHR001 and Tamika Surajpal SRJTAM001

a)

```r
dat$response <- apply(dat[, c("Y1", "Y2", "Y3")], 1, function (x){

  if (x[1] == 1) return("code-Alpha")
  if (x[2] == 1) return("code-Beta")
  if (x[3] == 1) return("code-Rho")
})

ggplot(dat, aes(x=X1, y=X2, color=response))+
         geom_point(size=2)+
         coord_fixed()+ # 1:1 aspect ratio
         labs(title = "Scatterplot of particles in feature space", x = "First coordinate (X1
         theme_minimal()
```

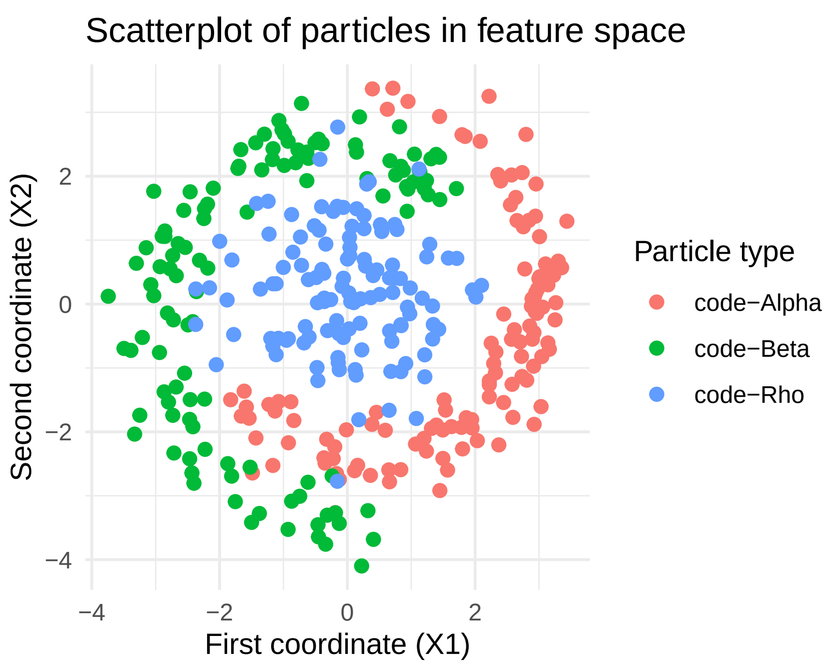


Figure 1: This is a scatter plot of the Collider data in the X1 and X2 feature space, with points colour-coded by true particle class (in red, in blue, in green), illustrating non-linear boundaries that motivate using a neural network for classification.

Figure 1 shows three distinct clusters corresponding to the particle types: `code-Alpha`, `code-Beta`, and `code-Rho`. While the classes are broadly separated — with `code-Alpha` particles primarily located on the right, `code-Beta` particles on the left, and `code-Rho` particles clustered near the center — the boundaries between these regions are clearly non-linear. This structure suggests that linear classifiers would struggle to accurately separate the classes.

Thus, using non-linear models such as a neural network is appropriate for this classification task, as they can capture the curved, complex boundaries observed in the feature space.

b)

```
softmax <- function(Z)
  {
  Z_shift <- Z - matrix(apply(Z, 2, max), nrow = 3, ncol = ncol(Z),byrow = TRUE) # Subtract

  expZ    <- exp(Z_shift)
```

```r
  denom   <- matrix(colSums(expZ),                   # column-wise sums...sum across 3 classes
                    nrow = 3, ncol = ncol(Z),        # convert to matrix for conformability
                    byrow = TRUE)

  expZ / denom
  }
```

c)

The function `calc_Ci` can be defined as follows:

$$\texttt{calc\_Ci}(y_{\text{true}}, y_{\text{pred}}) = \begin{cases} -\log(y_{\text{pred}}), & \text{if } y_{\text{true}} = 1, \\ 0, & \text{otherwise.} \end{cases}$$

In this function, (y_{true}) is assumed to be a scalar value that takes the value 0 or 1, representing the true class label. If (y_{true} = 1), the function computes the negative logarithm of (y_{pred}), which is the predicted probability for the true class. If (y_{true} = 0), the function simply returns 0.

```r
calc_Ci <- function(y_true, y_pred) {
  # y_true is assumed to be a scalar (either 0 or 1)
  if (y_true == 1) {
    return(-log(y_pred))
  } else {
    return(0)
  }
}
```

Explanation:

The reason it is advantageous to evaluate only the terms corresponding to (y_i = 1) is that the objective function, (C_i), only contributes a non-zero value when the true class label (y_{ij}) is 1. This means that for each data point, only one term in the summation is relevant for the calculation of the objective. Evaluating only the relevant term reduces unnecessary computations and improves numerical efficiency.

In practice, this method of selectively evaluating the terms corresponding to the true class labels (when (y_i = 1)) is beneficial because: - It avoids redundant calculations for terms where (y_{ij} = 0), which would contribute 0 to the sum. - It reduces the risk of numerical instability in floating-point operations, especially when working with very small values in the log function.

Thus, by focusing on the relevant terms, the computation becomes more efficient and numerically stable.

d)

```r
g <- function(Yhat, Y, eps = 1e-15) {
  # Yhat, Y : N × q matrices   (rows = obs, cols = classes)
  N <- nrow(Y)
  -sum( Y * log( pmax(Yhat, eps) ) ) / N
}
```

e)

The number of parameters \(N_{\text{params}} \) in an \((m, m)\)-Auto-Feature network with input dimension \(p \) and output dimension \(q \) is given by:

$$N_{\text{params}} = 2p^2 + 2p + 2pm + 2m + m^2 + mq + q.$$

In this expression, \(p \) represents the input dimension, \(m \) is the number of hidden nodes, and \(q \) is the output dimension. The terms in the formula correspond to the parameters in the various layers of the network, including weights and biases.

Number of parameters = $2p^2+2p+2pm+2m+m^2+mq+q$

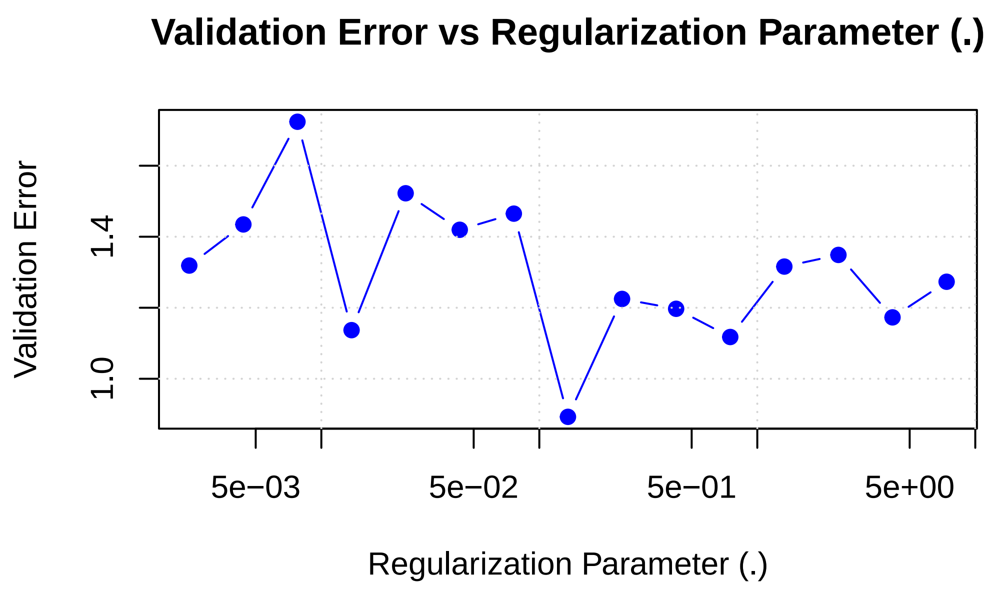
f)

```r
af_forward <- function(X, Y, theta, m, nu)
{
  N <- nrow(X)
  p <- ncol(X)
  q <- ncol(Y)

  index <- 1:(2*(p^2)) #W1 : p(p+p)
  W1 <- matrix(theta[index], nrow=p)

  index <- max(index)+1:(2*p) #b1 : (p+p)
  b1 <- theta[index]
  index <- max(index)+1:((2*p)*m) #W2 : (p+p)*m
  W2 <- matrix(theta[index], nrow=2*p)

  index <- max(index)+1:m #b2 : m
  b2 <- theta[index]

  index <- max(index)+1:(m*m) #W3 : (m*m)
  W3 <- matrix(theta[index], nrow=m)

  index <- max(index)+1:m #b3 : m
  b3 <- theta[index]

  index <- max(index)+1:(m*q) #W4 : (m*q)
  W4 <- matrix(theta[index], nrow=m)

  index <- max(index)+1:q #b4 : q
  b4 <- theta[index]

  #forward propagation
  H1 <- tanh( X  %*% W1 + matrix(b1, N, 2*p, TRUE) )       # aug-layer
  H2 <- tanh( H1 %*% W2 + matrix(b2, N, m, TRUE) )      # 2nd hidden
  H3 <- tanh(H2 %*% W3 + matrix(b3, N, m, TRUE))
  Z <- H3 %*% W4 + matrix(b4, N, q, TRUE) # logits

  #used the colsums in softmax but then transposed these probabilities
  P_3byN <- softmax(t(Z))   # t(Z) is q×N but q=3 here
  probs   <- t(P_3byN)

  #losses & objective
  loss <- g(probs, Y)                   # cross-entropy
  obj  <- loss + (nu / 2) * sum(theta^2)
  list(probs = probs, loss = loss, obj = obj)

}
```

g)

```
set.seed(2025)
```

```r
# Step 1: Split the data into training and validation sets (80%/20%)
n <- nrow(dat)
train_size <- floor(0.8 * n)
train_indices <- sample(1:n, train_size)
train_data <- dat[train_indices, ]
valid_data <- dat[-train_indices, ]

# Step 2: Prepare the training and validation datasets
X_train <- as.matrix(train_data[, 1:3])  # Input features
Y_train <- as.matrix(train_data[, 4:6])  # Response variables (one-hot encoded)

X_valid <- as.matrix(valid_data[, 1:3])  # Input features
Y_valid <- as.matrix(valid_data[, 4:6])  # Response variables (one-hot encoded)

# Step 3: Define the objective function with regularization
objective_fn <- function(theta, X, Y, m, nu) {
  result <- af_forward(X, Y, theta, m, nu)
  return(result$obj)
}

# Step 4: Grid search over regularization parameter nu
nu_values <- exp(seq(-6, 2, length.out = 15))
validation_errors <- numeric(length(nu_values))

for (i in 1:length(nu_values)) {
  nu <- nu_values[i]

  # Initial random theta
  p <- ncol(X_train)
  q <- ncol(Y_train)
  m <- 4
  npars <- 87
  theta_rand <- runif(npars, -1, 1)

  # Fit the model using optim() to minimize the objective function
  fit <- optim(theta_rand, objective_fn, X = X_train, Y = Y_train, m = 4, nu = nu)

  # Get the predicted probabilities for validation set
  Yhat_valid <- af_forward(X_valid, Y_valid, fit$par, m = 4, nu = nu)$probs

  # Compute the validation error
  validation_errors[i] <- g(Yhat_valid, Y_valid)
}

# Step 5: Plot validation error vs nu
plot(nu_values, validation_errors, type = "b", col = "blue", pch = 19, asp = 1, log="x",
     xlab = "Regularization Parameter ( )", ylab = "Validation Error",
     main = "Validation Error vs Regularization Parameter ( )")
grid()
```



```r
# Step 6: Choose the optimal regularization level ( )
optimal_nu <- nu_values[which.min(validation_errors)]
```

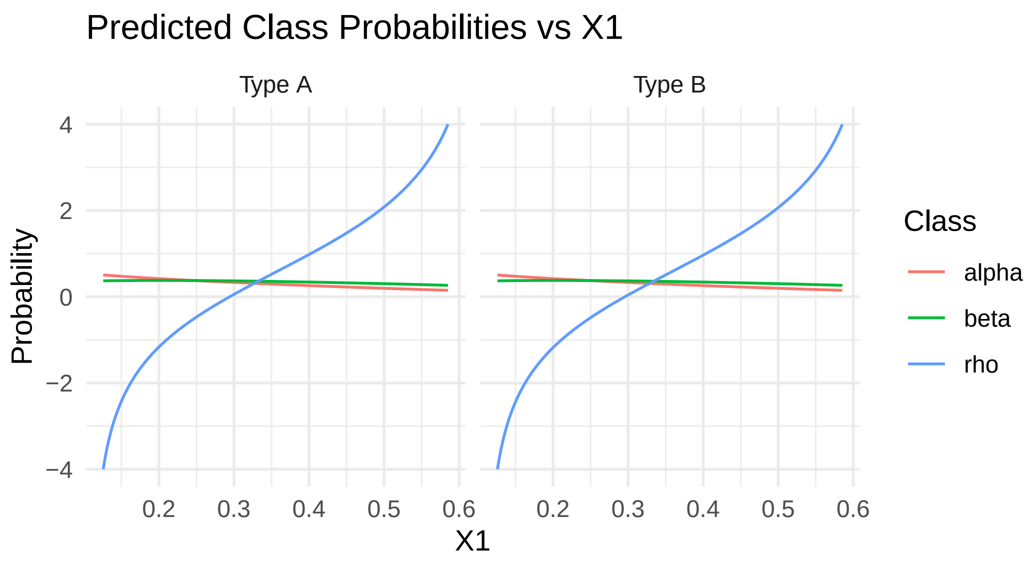
The optimal regularization parameter is 0.1353 as it minimizes the validation error.

h)

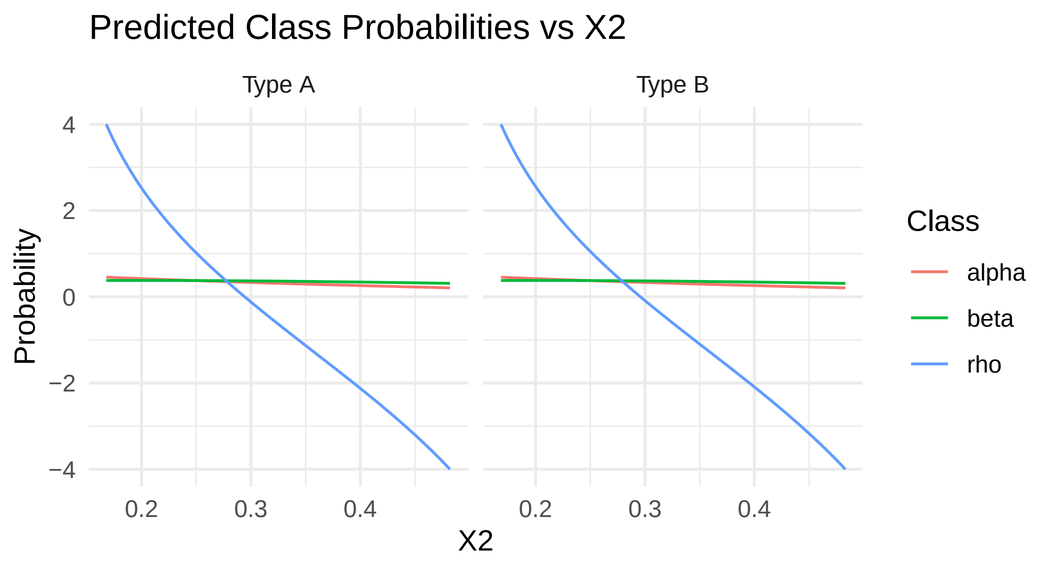
```r
m <- 4
best_nu <- exp(-2)
nu <- best_nu
```

```r
# Re-train model at best nu
obj_pen_best <- function(pars) {
  af_forward(X_train, Y_train, pars, m, nu)$obj
}

theta_rand <- runif(npars, -1, 1)  # Reinitialize random parameters
res_opt_best <- nlm(obj_pen_best, p = theta_rand, iterlim = 1000)
theta_best <- res_opt_best$estimate


# Plot response curves by varying X1 and X2 separately

# Helper function to predict probability curves
predict_curve <- function(var_seq, varname, fixed_X2 = 0, fixed_X3 = 0, pars, m) {
  n <- length(var_seq)
  input <- matrix(0, nrow = n, ncol = 3)
  colnames(input) <- c("X1", "X2", "X3")

  input[, "X1"] <- if (varname == "X1") var_seq else fixed_X2
  input[, "X2"] <- if (varname == "X2") var_seq else fixed_X2
  input[, "X3"] <- fixed_X3

  q <- 3

  preds <- af_forward(input, Y = matrix(0, nrow=n, ncol=q), pars, m, nu=0)$probs

  out <- as.data.frame(preds)
  colnames(out) <- c("alpha", "beta", "rho")
  out[[varname]] <- var_seq

  return(out)
}

# Create sequences
X_seq <- seq(-4, 4, length.out = 100)

# Response curves for Detector Type A (X3=1) and Type B (X3=0)

curve_X1_A <- predict_curve(X_seq, "X1", fixed_X2=0, fixed_X3=1, theta_best, m)
curve_X1_B <- predict_curve(X_seq, "X1", fixed_X2=0, fixed_X3=0, theta_best, m)

curve_X2_A <- predict_curve(X_seq, "X2", fixed_X2=0, fixed_X3=1, theta_best, m)
```

```r
curve_X2_B <- predict_curve(X_seq, "X2", fixed_X2=0, fixed_X3=0, theta_best, m)

# Helper to prepare data
prepare_plot_data <- function(curve_data, varname, type_label) {
  df <- as.data.frame(curve_data)
  colnames(df) <- c(varname, "alpha", "beta", "rho")
  df$Detector <- type_label
  df <- pivot_longer(df, cols = c("alpha", "beta", "rho"),
                     names_to = "Class", values_to = "Probability")
  return(df)
}

plot_data_X1 <- bind_rows(
  prepare_plot_data(curve_X1_A, "X1", "Type A"),
  prepare_plot_data(curve_X1_B, "X1", "Type B")
)

ggplot(plot_data_X1, aes(x = X1, y = Probability, color = Class)) +
  geom_line() +
  facet_wrap(~ Detector) +
  labs(title = "Predicted Class Probabilities vs X1",
       x = "X1", y = "Probability") +
  theme_minimal() +
  theme(aspect.ratio = 1)
```



```r
plot_data_X2 <- bind_rows(
  prepare_plot_data(curve_X2_A, "X2", "Type A"),
  prepare_plot_data(curve_X2_B, "X2", "Type B")
)

ggplot(plot_data_X2, aes(x = X2, y = Probability, color = Class)) +
  geom_line() +
  facet_wrap(~ Detector) +
  labs(title = "Predicted Class Probabilities vs X2",
       x = "X2", y = "Probability") +
  theme_minimal() +
  theme(aspect.ratio = 1)
```



i)

One practical advantage of using an AutoFeature (AF) network over a standard feedforward neural network is that AF networks automatically discover important transformations of the input variables, such as quadratic terms and interactions, without requiring manual feature engineering. In our Prac, we saw that the AF network easily captured structured nonlinear relationships between X1, X2, and the class probabilities (, ,) — for example, the probability of class showed a clear nonlinear upward trend with X1, even though no complex hidden layers were manually designed. This shows that the AF network can model nonlinearity efficiently with simple transformations.

Another advantage is interpretability. Because the AF network was designed to learn specific types of feature mappings (rather than arbitrary complex patterns), we could directly observe how the inputs influenced the outputs through the plotted response curves. For instance, the nearly linear but slightly sloped trends for and made it clear how X1 and X2 affected the probabilities. In contrast, a standard feedforward network would have created highly entangled hidden features, making it difficult to extract such insights. Finally, by constraining the feature space, the AF network also helped prevent overfitting, which is especially important when working with limited training data, as we had in this Prac.