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

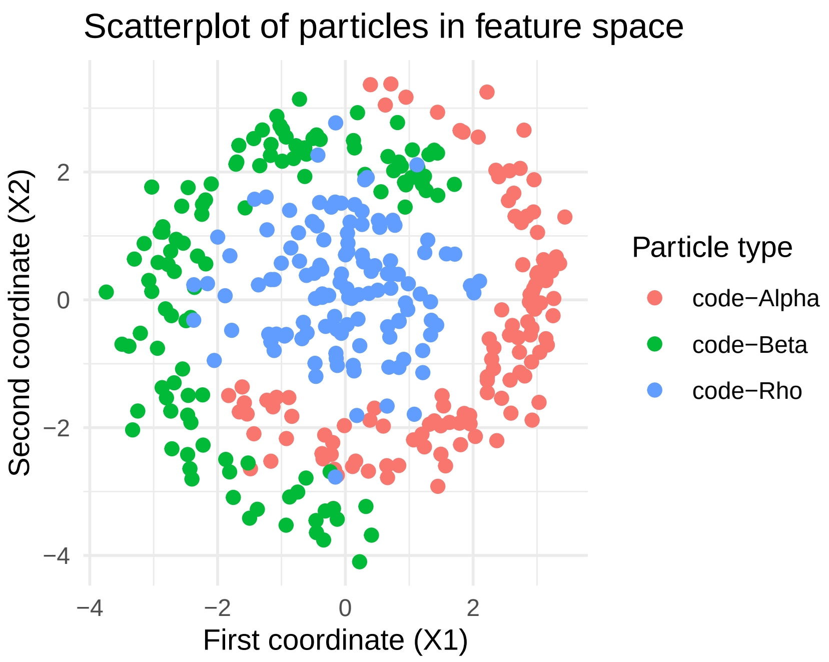

Figure 1: This is a scatter plot of the Collider data in the X1 and X2 feature space, with points colour-coded by true particle class, illustrating non-linear boundaries that motivate using a neural network for classification.

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)

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

    # Subtract the column max for numerical stability (to avoid computational
    # overflow when exponentiating)
    expZ <- exp(Z_shift)
    denom <- matrix(colSums(expZ), nrow = 3, ncol = ncol(Z), byrow = TRUE)
    # column-wise sums...sum across 3 classes convert to matrix for
    # conformability
    expZ/denom
}
```

c)

We can define the contribution of a single observation i to the cross-entropy loss, C_i, using the following case-based function:

$$C_i = \begin{cases} -\log(\hat{y}_{ij}) & \text{if } y_{ij} = 1, \\ 0 & \text{otherwise.} \end{cases}$$

It is advantageous, for numerical purposes, to evaluate only the term corresponding to the class where $y_{ij} = 1$. This is because for all other classes, $y_{ij} = 0$, and their contributions to the sum are therefore exactly zero. Avoiding unnecessary computation of terms like $0 \times \log(\hat{y}_{ij})$ improves numerical stability, reduces computational cost, and prevents possible undefined operations such as taking the logarithm of zero.

d)

```r
g <- function(Yhat, Y, eps = 1e-15) {
    # Yhat, Y : N × q matrices (rows = observations, columns = classes)
    N <- nrow(Y)
    -sum(Y * log(pmax(Yhat, eps)))/N
    # pmax() replaces any element of Yhat that is smaller than eps with eps
    # #ensures no value passed to log() is <= zero
}
```

e)

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

$$n_{\text{pars}} = 2p^2 + 2p + 2pm + 2m + m^2 + mq + q.$$

f)

```r
# X: input matrix (N x p) Y: output matrix (N x q) theta: parameter vector with
# all weights and biases m: number of nodes on hidden layer v: regularisation
# parameter
af_forward <- function(X, Y, theta, m, v) {
    N <- nrow(X)
    p <- ncol(X)
    q <- ncol(Y)

    # Populate weight-matrix and bias vectors by unpacking theta:
    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 output
    H2 <- tanh(H1 %*% W2 + matrix(b2, N, m, TRUE))  # 1st hidden layer output
    H3 <- tanh(H2 %*% W3 + matrix(b3, N, m, TRUE))  # 2nd hidden layer output
    Z <- H3 %*% W4 + matrix(b4, N, q, TRUE)  # final layer to get logits

    # apply softmax across logits
    P_3byN <- softmax(t(Z))
    # temporarily transpose because softmax expects input where columns are
    # different samples
    probs <- t(P_3byN)

    # losses & objective
    loss <- g(probs, Y)  # cross-entropy
    obj <- loss + (v/2) * sum(theta^2)  # L2 regularisation
    list(probs = probs, loss = loss, obj = obj)

}
```

g)

```
set.seed(2025)
m <- 4
# 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
v = 0.01
obj_pen <- function(theta) {
    result <- af_forward(X_train, Y_train, theta, m, v)
    return(result$obj)
}

# Initial random theta
p <- ncol(X_train)
q <- ncol(Y_train)
m <- 4
npars <- 2 * p^2 + 2 * p + 2 * p * m + 2 * m + m^2 + m * q + q
theta_rand = runif(npars, -1, 1)

res_opt = nlm(obj_pen, theta_rand, iterlim = 1000)

# Step 4: Grid search over regularization parameter nu
n_v <- 25
validation_errors = rep(NA, n_v)
v_values <- exp(seq(-10, 1, length.out = n_v))
# validation_errors <- numeric(length(v_values))
```

```r
for (i in 1:n_v) {
    v <- v_values[i]
    res_opt = nlm(obj_pen, theta_rand, iterlim = 1000)

    res_val = af_forward(X_valid, Y_valid, res_opt$estimate, m, 0)
    validation_errors[i] = res_val$obj
}

best_i <- which.min(validation_errors)
best_v <- v_values[best_i]

# Step 5: Plot validation error vs v
plot(v_values, validation_errors, type = "b", log = "x", pch = 16, xlab = expression(nu),
    ylab = "Validation Cross-Entropy Loss", main = "Validation Loss vs Regularization",
    asp = 1)
abline(v = best_v, col = "red", lty = 2)
points(best_v, validation_errors[best_i], col = "red", pch = 16)
legend("topright", legend = paste0("Chosen v=", signif(best_v, 3)), col = "red",
    lty = 2, pch = 16, bty = "n")
```

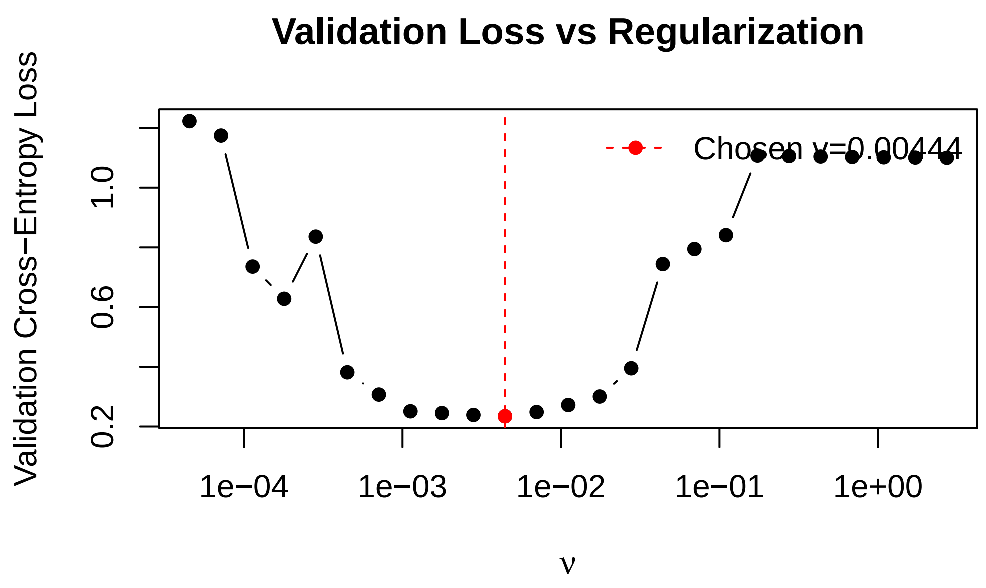

Figure 2: Validation cross-entropy loss on the held-out set plotted against the L2-regularization parameter v(log-scaled) for an AF-network with m=4. The red dashed line marks the chosen optimal v. We use a logarithmic scale because v ranges over several orders of magnitude; the log-scale stretches out the small v region so that the classic U-shaped trade-off between under- and over-regularization becomes visible.

The optimal regularization parameter is 0.0044 as it minimizes the validation error.

h)

```r
# Fit the final model at optimal regularization
m <- 4
p <- ncol(X_train)
q <- ncol(Y_train)
npars <- 2 * p^2 + 2 * p + 2 * p * m + 2 * m + m^2 + m * q + q

theta_init <- runif(npars, -1, 1)
obj_pen_best <- function(theta) {
    af_forward(X_train, Y_train, theta, m, best_v)$obj
}
res_opt_best <- nlm(obj_pen_best, theta_init, iterlim = 1000)
theta_best <- res_opt_best$estimate

# Prepare medians and sequences
X1_med <- median(dat$X1)
X2_med <- median(dat$X2)
X1_seq <- seq(min(dat$X1), max(dat$X1), length.out = 200)
X2_seq <- seq(min(dat$X2), max(dat$X2), length.out = 200)

# Build grids for sweeping X1 and X2
grid_X1 <- expand.grid(X1 = X1_seq, X2 = X2_med, X3 = c(0, 1))
grid_X2 <- expand.grid(X1 = X1_med, X2 = X2_seq, X3 = c(0, 1))

# Get predicted probabilities (v = 0 for final preds)
probs_X1 <- af_forward(X = as.matrix(grid_X1), Y = matrix(0, nrow(grid_X1), 3), theta = theta_best,
    m = m, v = 0)$probs

probs_X2 <- af_forward(X = as.matrix(grid_X2), Y = matrix(0, nrow(grid_X2), 3), theta = theta_best,
    m = m, v = 0)$probs

# Attach class names and reshape to long form
df1 <- cbind(grid_X1, setNames(as.data.frame(probs_X1), c("alpha", "beta", "rho")))

df1$Detector <- factor(df1$X3, levels = c(0, 1), labels = c("Type B", "Type A"))

df1_long <- pivot_longer(df1, cols = c("alpha", "beta", "rho"), names_to = "Class",
    values_to = "Probability")

df2 <- cbind(grid_X2, setNames(as.data.frame(probs_X2), c("alpha", "beta", "rho")))

df2$Detector <- factor(df2$X3, levels = c(0, 1), labels = c("Type B", "Type A"))

df2_long <- pivot_longer(df2, cols = c("alpha", "beta", "rho"), names_to = "Class",
    values_to = "Probability")

# Plot response curves for X1
ggplot(df1_long, aes(x = X1, y = Probability, color = Class)) + geom_line(size = 1) +
    facet_wrap(~Detector) + labs(title = "Response Curves: P(class) vs X1 by Detector Type",
    x = "X1", y = "Predicted Probability", color = "Class") + theme_minimal() + theme(aspect.ratio = 1)
```

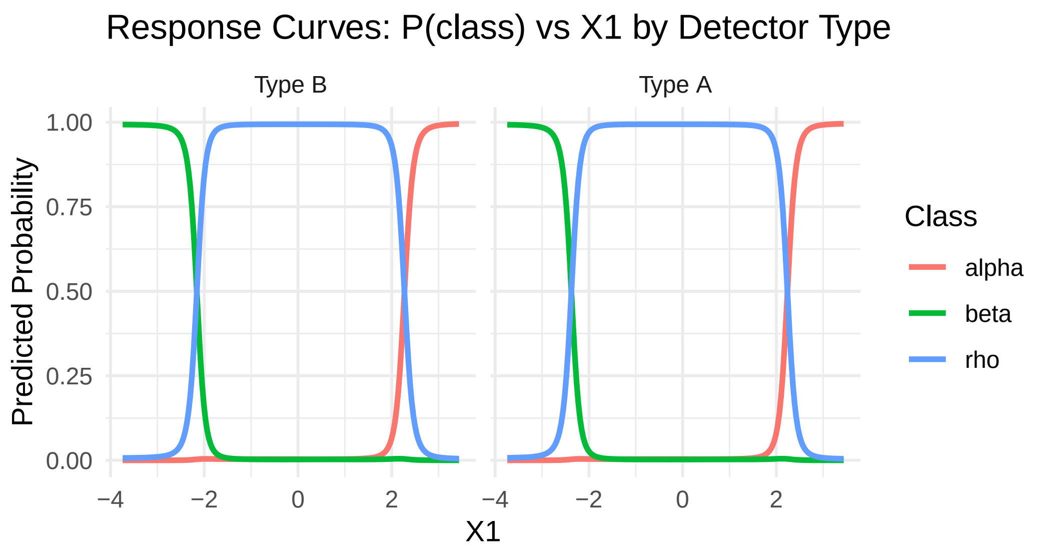

Figure 3: This figure shows two faceted line plots showing how the neural network's predicted class probabilities vary as X1 changes, separately for Detector Type A (X3 = 1) and Type B (X3 = 0).

```
# Plot response curves for X2
ggplot(df2_long, aes(x = X2, y = Probability, color = Class)) + geom_line(size = 1) +
    facet_wrap(~Detector) + labs(title = "Response Curves: P(class) vs X2 by Detector Type",
    x = "X2", y = "Predicted Probability", color = "Class") + theme_minimal() + theme(aspect.ratio = 1)
```

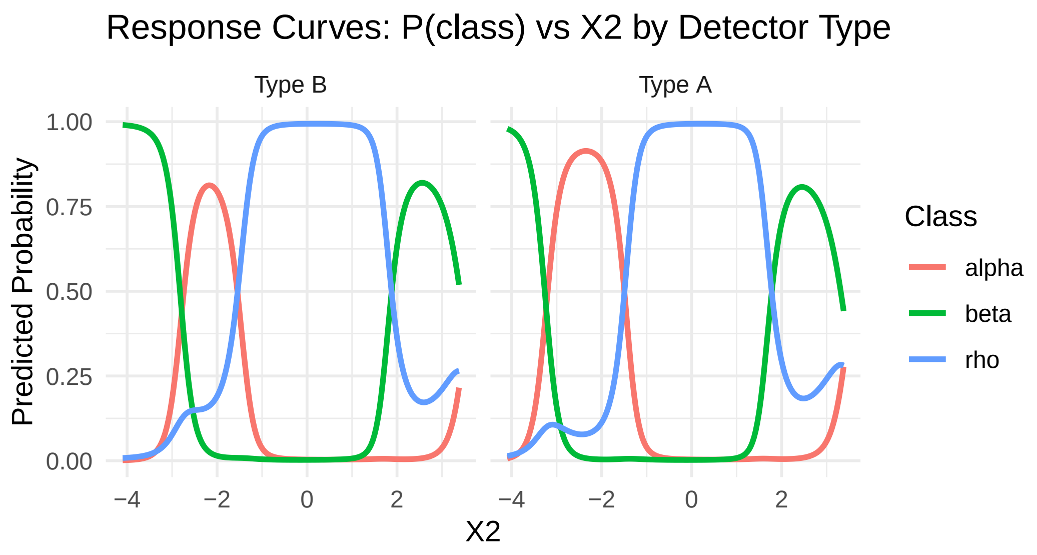

Figure 4: This figure shows two faceted line plots showing how the neural network's predicted class probabilities vary as X2 changes, separately for Detector Type A (X3 = 1) and Type B (X3 = 0).

i)

One practical advantage of using an AF network over a standard feed forward 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 α, β, ρ — for example, the probability of class ρ 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 key benefit is interpretability. Because the AF-network augments the feature space in a controlled way, we can directly read off how each input drives each class probability by plotting response curves. Part (h) showed that the entire family of red, green and blue curves for Detector A lies almost perfectly on top of the matching curves for Detector B. Therefore, flipping X3 from 0->1 produces no shift in P(α), P(β) or P(ρ). This tells us that the model is truly exploiting the nonlinear structure in X ,X . And once those two coordinates are accounted for, the detector flag carries no extra information and is effectively ignored by the fitted network. By, constraining the feature space to just those learned augmentations, the AF-network also guards against over-fitting which is crucial when you have only a few hundred labelled particles.