Machine Learning exam help

Manfred Clase

Contents

1 Lab 1 **2**
1.1 Assignment 1: Handwritten Digit Recognition (KNN) 2
1.2 Assignment 2: Linear & Ridge Regression 6
1.3 Assignment 3: Logistic Regression & Basis Expansion 10

2 Lab 2 **12**
2.1 Assignment 1: Explicit Regularization (LASSO & Ridge) 12
2.2 Assignment 2: Decision Trees & Logistic Regression (Bank Marketing) 14
2.3 Assignment 3: PCA & Implicit Regularization (Crime Data) 17

3 Lab 3 **20**
3.1 Assignment 2: Kernel Methods (Temperature Prediction) 20
3.2 Assignment 3: Support Vector Machines (SVM) 22
3.3 Assignment 4: Neural Networks (Sine Regression) 24

4 Exam 2024-08-28 **26**
4.1 Assignment 1: Trees, Logistic Regression & MLE 26
4.2 Assignment 2: Neural Networks (Backprop & Dropout) 29

5 Exam 2024-01-12 **32**
5.1 Assignment 1: Lasso Regression & Custom Optimization 32
5.2 Assignment 2: Neural Networks (Dropout) & Perceptron 35

6 Exam 2023-03-17 **38**
6.1 Assignment 1: KNN, PCA Optimization 38
6.2 Assignment 2: SVM Kernel Methods 41

7 Exam 2022-03-18 **43**
7.1 Assignment 1: PCA, Logistic Regression & Custom Optimization 43
7.2 Assignment 2: Kernel Density & Neural Networks 46

8 Exam 2025-03-20 **49**
8.1 1.1 LASSO Regression for Area Prediction 49
8.2 1.2 Logistic Regression for Diagnosis Classification 50
8.3 1.3 Linear Classification with Hinge Loss 51
8.4 Assignment 2. Kernel Models 53
8.5 Assignment 3: Neural Networks 55

9 Exam 2025-01-17 **56**
9.1 Assignment 1 (10p) 56
9.2 Assignment 2: Kernel Models (6p) 59
9.3 Assignment 3: Neural Networks (4p) 62

10 CUSTOM PROBLEMS: LDA, CNN ADVANCED DIMENSIONALITY REDUCTION **63**
10.1 Extra Assignment: LDA, CNN & Advanced Dim. Reduction 63

11 Theory Gaps (From Slides) **64**
11.1 Handling Loss Matrices 64
11.2 Log Reg probabilsitic equation 65

12 Exam 2022, hold out, decision tree **65**

13 Exam 2023-01-13, basis function expansion, logreg, decision trees 67

14 Formulas and Cost Functions from TDDE01 Exams 69
14.1 1. Probabilistic and Prediction Models 69
14.2 2. Cost and Objective Functions 70
14.3 3. Density Estimation and Kernel Methods 70
14.4 4. Algorithm Updates 70

Lab 1

Assignment 1: Handwritten Digit Recognition (KNN)

Problem Statement:
Data: *optdigits.csv.* ***Target:*** *Last col (digit 0-9).*

1. *Import data. Partition: 50% Train, 25% Valid, 25% Test.*
2. *Fit 30-NN classifier. Report Confusion Matrices Misclassification Errors for Train/Test.*
3. *Find and visualize the 2 ”easiest” and 3 ”hardest” digits (specifically digit ”8”) to classify based on probability.*
4. *Fit KNN for $K = 1 \ldots 30$. Plot Train vs Validation error. Find optimal K. Estimate Test Error for optimal K.*
5. *Compute Cross-Entropy Error on validation data for $K = 1 \ldots 30$. Plot and compare optimal K.*

```
#setup and data import
#load data, header=FALSE tells R the first row is data, not labels
data <- read.csv("optdigits.csv", header = FALSE)
#the last column (65) contains the actual digit (0-9)
#rename to "label" for clarity
colnames(data)[65] <- "label"
#Important: Convvert the target variable to a factor
#tells R we are doing classification (categories), not Regression (numbers)
data$label <- as.factor(data$label)
#=========data partitioning ====
#count total numbers of rows (n)
n <- nrow(data)
#create training set 50%
set.seed(12345)
#random 50% of row indices
id_train <- sample(1:n, floor(n * 0.50))
#create the training dataframe using these indices
train_data <- data[id_train, ]
# step B, create validation set (25%)
#first, identify which rows are left (total indices minus Training indices)
id_remaining <- setdiff(1:n, id_train)
#set seed for random
set.seed(12345)
#sample 25% of the total n from the remaining indices
id_valid <- sample(id_remaining, floor(n * 0.25))
#create validation dataframe
valid_data <- sample(id_remaining, floor(n * 0.25))
#create validation dataframe
valid_data <- data[id_valid, ]
#step C create create test set (25%)
#the test set is whats left (remaining indices minus validation indices)
id_test <- setdiff(id_remaining, id_valid)
test_data <- data[id_test, ]

library (kknn)
#======== 1.2 k-nn Classifier (k=30)
#predict on test data!!
#we use train_data as the "knowledge base" and test_data as the query points
knn_test_model <- kknn(formula = label ~ ., train = train_data, test = test_data,
```

```r
                       k = 30, kernel = "rectangular")
#extract the predicted class labels
pred_test <- fitted(knn_test_model)
#compute confusion matrix (rows = predicted, cols = actual)
cm_test <- table(pred_test, test_data$label)
#Missclassification Error
#Accuracy = (Sum of diagonal elements) / Total elements
#error = 1 - Accuracy
acc_test <- sum(diag(cm_test)) / sum(cm_test)
err_test <- 1 - acc_test
print(err_test)
#predict on training data
knn_train_model <- kknn(formula = label ~ ., train = train_data,
                        test = train_data, k=30, kernel = "rectangular")
pred_train <- fitted(knn_train_model)
#confusion matrix (training)
cm_train <- table(pred_train, train_data$label)
print(cm_train)
#missclassification error on training:
acc_train <- sum(diag(cm_train)) / sum(cm_train)
err_train <- 1 - acc_train
print(err_train)

# ============================================================
# Task 1.3: Visualizing the Easiest and Hardest '8's
# ============================================================
library(kknn)
# 1. Fit the model: Training Data predicting Training Data
#    We need this to get the probabilities for the training set.
model_train <- kknn(label ~ .,
                    train = train_data,
                    test = train_data,
                    k = 30,
                    kernel = "rectangular")
# 2. Extract probabilities
#    prob_matrix rows correspond to the rows in train_data
prob_matrix <- model_train$prob
# 3. Filter: Find rows where the ACTUAL label is "8"
idx_8 <- which(train_data$label == "8")
probs_for_8 <- prob_matrix[idx_8, "8"]
# 4. Map Indices: This is the crucial step to avoid "subscript out of bounds"
#    We assign the original row numbers as names to the probability vector
names(probs_for_8) <- idx_8
# 5. Sort to find Easiest (High Prob) and Hardest (Low Prob)
sorted_probs_8 <- sort(probs_for_8, decreasing = FALSE) # Low to High
# Hardest = Lowest probability (First 3 in sorted list)
hardest_indices <- as.numeric(names(head(sorted_probs_8, 3)))
hardest_probs   <- head(sorted_probs_8, 3)
# Easiest = Highest probability (Last 2 in sorted list)
easiest_indices <- as.numeric(names(tail(sorted_probs_8, 2)))
easiest_probs   <- tail(sorted_probs_8, 2)
# --- Visualization Function ---
plot_digit <- function(row_idx, prob_value, type) {
  # Extract the 64 pixels (columns 1 to 64)
  pixel_vector <- as.numeric(train_data[row_idx, 1:64])
  # Reshape to 8x8
  digit_matrix <- matrix(pixel_vector, nrow = 8, ncol = 8, byrow = TRUE)
  # Flip the matrix rows so the image isn't upside down in heatmap()
  digit_matrix_flipped <- digit_matrix[8:1, ]
  # Plot
  heatmap(digit_matrix_flipped,
          Colv = NA, Rowv = NA, scale = "none",
          col = grey.colors(255, start = 1, end = 0), # White to Black
          main = paste(type, "8 (Row:", row_idx, ")\nProb:", round(prob_value, 4)))
}
# --- Execute Plotting ---
# Setup grid: 2 rows, 3 columns (to fit 5 images nicely)
par(mfrow = c(2, 3))
# Plot 3 Hardest
for (i in 1:3) {
  plot_digit(hardest_indices[i], hardest_probs[i], "Hardest")
}
# Plot 2 Easiest
for (i in 1:2) {
  plot_digit(easiest_indices[i], easiest_probs[i], "Easiest")
}
# Reset layout
par(mfrow = c(1, 1))

# ============================================================
# Task 1.4: Optimization of K (1 to 30)
# ============================================================
# 1. Initialize vectors to store misclassification errors
k_values <- 1:30
train_errors <- numeric(30)
valid_errors <- numeric(30)
# 2. Loop through K = 1 to 30
for (k in k_values) {
  # --- A. Calculate Training Error ---
  # We use train_data as both training and test input to check how well it memorizes data
  model_train <- kknn(formula = label ~ .,
                      train = train_data,
                      test = train_data,
                      k = k,
                      kernel = "rectangular")
  # Get predictions and compute error
  pred_train <- fitted(model_train)
  cm_train <- table(pred_train, train_data$label)
  train_errors[k] <- 1 - sum(diag(cm_train)) / sum(cm_train)
  # --- B. Calculate Validation Error ---
  # We use train_data to teach, and valid_data to test
  model_valid <- kknn(formula = label ~ .,
                      train = train_data,
                      test = valid_data,
                      k = k,
                      kernel = "rectangular")
  pred_valid <- fitted(model_valid)
  cm_valid <- table(pred_valid, valid_data$label)
  valid_errors[k] <- 1 - sum(diag(cm_valid)) / sum(cm_valid)
}
# 3. Plot the dependencies
# Plot Validation Error (Red)
plot(k_values, valid_errors, type = "b", col = "red", ylim = c(0, 0.06),
     xlab = "K (Number of Neighbors)", ylab = "Misclassification Error",
     main = "k-NN Model Complexity: Error vs K")
# Add Training Error (Blue)
lines(k_values, train_errors, type = "b", col = "blue")
# Add Legend
legend("bottomright", legend = c("Validation Error", "Training Error"),
       col = c("red", "blue"), lty = 1, pch = 1)
# 4. Find the Optimal K
# This finds the index (K) where valid_error is minimum
optimal_k <- which.min(valid_errors)
print(paste("Optimal K based on Validation Error:", optimal_k))
```

```
# ============================================================
# Final Step: Estimate Test Error for the Optimal K
# ============================================================
# Now we run the model one last time using the optimal K on the TEST set
model_test_opt <- kknn(formula = label ~ .,
                       train = train_data,
                       test = test_data,
                       k = optimal_k,
                       kernel = "rectangular")
pred_test_opt <- fitted(model_test_opt)
cm_test_opt <- table(pred_test_opt, test_data$label)
test_error_opt <- 1 - sum(diag(cm_test_opt)) / sum(cm_test_opt)
# Report all three errors for comparison
print(paste("Training Error (at K=", optimal_k, "):", train_errors[optimal_k]))
print(paste("Validation Error (at K=", optimal_k, "):", valid_errors[optimal_k]))
print(paste("Test Error (at K=", optimal_k, "):", test_error_opt))

# ============================================================
# Task 1.5: Cross-Entropy Error vs K
# ============================================================
# 1. Initialize vectors
k_values <- 1:30
valid_cross_entropy <- numeric(30)
# 2. Loop through K
for (k in k_values) {
  # Train k-NN model
  model <- kknn(formula = label ~ .,
                train = train_data,
                test = valid_data,
                k = k,
                kernel = "rectangular")
  # A. Get the probability matrix
  prob_matrix <- model$prob
  # B. Get the correct labels
  correct_labels <- valid_data$label
  # C. ROBUST EXTRACTION:
  # instead of relying on names directly, find which column index
  # corresponds to the true label for each row.
  # This avoids the "subscript out of bounds" error.
  col_indices <- match(as.character(correct_labels), colnames(prob_matrix))
  # Extract the probability for the correct class using the indices
  # prob_matrix[row_index, column_index]
  prob_correct_class <- prob_matrix[cbind(1:nrow(valid_data), col_indices)]
  # D. Compute Cross-Entropy
  # Formula: - Sum( log( probability_of_correct_class ) )
  # We add the negative sign because we want "Error" (lower is better).
  valid_cross_entropy[k] <- -sum(log(prob_correct_class + 1e-15))
}
# Check the actual range of your values first to see how high they go
print(range(valid_cross_entropy))
# Plot with automatic y-limits (remove ylim argument entirely)
plot(k_values, valid_cross_entropy, type = "b", col = "darkgreen",
     xlab = "K (Number of Neighbors)", ylab = "Cross-Entropy Error",
     main = "Validation Error: Cross-Entropy vs K")
# 4. Find the new Optimal K
optimal_k_ce <- which.min(valid_cross_entropy)
print(paste("Optimal K based on Cross-Entropy:", optimal_k_ce))
```

2. 30-NN Quality & Matrices:

- **Confusion Matrix:** Rows = Predicted, Cols = True. High values on the *diagonal* indicate correct predictions. Off-diagonal values show specific confusions (e.g., predicting 9 when true label is 4).

- **Digit Quality:** Some digits are topologically similar (e.g., 1 vs 7, 3 vs 8, 9 vs 4). These pairs typically have higher misclassification rates than unique shapes like 0 or 6.

3. Easiest vs. Hardest Digits:

- **Easiest ($P(Class) \approx 1$):** The query point is surrounded entirely by neighbors of the same class. Visually, these are "prototypical" 8s with clear loops and vertical alignment.
- **Hardest ($P(Class) \ll 1$):** The query point has very few neighbors of the correct class (mostly surrounded by other digits). Visually, these 8s are likely distorted, broken, or resemble other digits (e.g., a loop is unclosed, looking like a 3).

4. Model Complexity K (Bias-Variance):

- **Small K (e.g., $K = 1$): High Complexity**. Low Bias, High Variance. The boundary is jagged and fits noise. Training error is 0 (overfitting), but Validation error is high.
- **Large K (e.g., $K = 30$): Low Complexity**. High Bias, Low Variance. The boundary is smooth. Training error increases (underfitting), Validation error starts high, drops, then rises again.
- **Optimal K:** The minimum point of the Validation Error curve.

5. Cross-Entropy vs. Misclassification Error:

- **Why Cross-Entropy?** Misclassification error is *discrete* (1 or 0)—it doesn't care if the model was 51% sure or 99% sure.
- **Benefit:** Cross-entropy is *continuous* and penalizes low confidence. If the model predicts the correct class but with low probability (e.g., 0.51), Cross-Entropy penalizes this more than if probability was 0.99. It encourages the model to be both *correct* and *confident*.

Assignment 2: Linear & Ridge Regression

Problem Statement:
Data: *parkinsons.csv.* ***Target:*** *motor_UPDRS.*

1. ***Setup:*** *Load data. Remove `subject.`, `age`, `sex`, `test_time`, `total_UPDRS`. Scale data. Partition 60/40.*
2. ***Linear Regression:*** *Fit a linear model (no intercept) to predict `motor_UPDRS`. Report Train/Test MSE.*
3. ***Manual Ridge:*** *Implement functions for:*
   - *`LogLikelihood`: Gaussian log-likelihood.*
   - *`Ridge`: Cost function ($-LL + \lambda \sum \theta^2$).*
   - *`RidgeOpt`: Optimize parameters using `optim(method="BFGS")`.*
   - *`DF`: Calculate Degrees of Freedom ($tr(H_\lambda)$).*
4. ***Evaluation:*** *Run optimization for $\lambda \in \{1, 100, 1000\}$. Report Train/Test MSE and DF for each.*

```r
# ============================================================
# ssignment 2.1: Data Setup
# ============================================================
# 1. Load Data
raw_data <- read.csv("parkinsons.csv")
# 2. Select ONLY the relevant columns
# We remove cols 1-6 (subject, age, sex, test_time, motor_UPDRS, total_UPDRS)
# BUT we need to keep motor_UPDRS as our target!
# Usually, motor_UPDRS is column 5 and total_UPDRS is column 6.
# Let's verify by name to be safe.
# We want 'motor_UPDRS' + all the voice features (Jitter, Shimmer, etc.)
# We explicitly REMOVE: subject., age, sex, test_time, total_UPDRS
vars_to_remove <- c("subject.", "age", "sex", "test_time", "total_UPDRS")
data_clean <- raw_data[, !(names(raw_data) %in% vars_to_remove)]
# 3. Scale the data
data_scaled <- as.data.frame(scale(data_clean))
# 4. Partition (60% Train, 40% Test)
n <- nrow(data_scaled)
set.seed(12345)
id_train <- sample(1:n, floor(n * 0.60))
train_data <- data_scaled[id_train, ]
```

```r
test_data  <- data_scaled[-id_train, ]
# 5. Create Matrices for Ridge (Now without the cheat variables)
X_train <- as.matrix(train_data[, colnames(train_data) != "motor_UPDRS"])
y_train <- train_data$motor_UPDRS
X_test <- as.matrix(test_data[, colnames(test_data) != "motor_UPDRS"])
y_test <- test_data$motor_UPDRS

#========================
#ASSIGNMENT 2.2 LINEAR REGRESSION NO INTERCEPT (-1 = no intercept)
 #fit the model
lin_model <- lm(motor_UPDRS ~ . - 1, data = train_data)
#print summary to find significant variables
print(summary(lin_model))
#compute MSE
compute_mse <- function(model, dataset, target_col){
  #predict
  predictions <- predict(model, dataset)
  #actual
  actuals <- dataset[[target_col]]
  #mse is mean of (prediction - actual)^2
  mean((predictions - actuals)^2)
}
mse_train <- compute_mse(lin_model, train_data, "motor_UPDRS")
mse_test <- compute_mse(lin_model, test_data, "motor_UPDRS")
print(paste("Training MSE:", mse_train))
print(paste("Test MSE: ", mse_test))

#====================
#Task 2.3 Implementing Ridge Regression functions
#====================
#A. LogLikelihood Function
#calculates the log prob of the data given the model parameters
#Input:
#theta: the weights (coefficients) for the features
#sigma: the standard deviation of the error (noise)
#X: the matrix of feature data (traniing data).
#y: the vector of target values (motor_UPDRS)
LogLikelihood <- function(theta, sigma, X, y){
  n <- length(y)
  #calculate prediction y = X * theta
  y_pred <- X %*% theta
  #calculate sum of Squared Errors (SSE)
  sse <- sum((y - y_pred)^2)
  #log likelikhood formula for normal distribution
  #the formula is derived from the Normal Probability Density function
  #Log(L) = -(n/2)*log(2*pi*sigma^2) - (1/2*sigma^2)) * SSE
  log_lik <- -(n/2) * log(2*pi*sigma^2) - (1/(2*sigma^2))*sse
  return(log_lik)
}
#B. RIDGE FUNCTION ============
#this is the cost function we want to MINIMIZE
#it calculates (Minus LogLikelihood) * (Ridge penalty)
#Input:
#par: A single vector containing BOTH theta nad sigma
#lambda: the penalty strength scalar
Ridge <- function(par, X, y, lambda) {
  #unluck parameters
  #the last number in "par" is sigma, the rest are theta
  p <- ncol(X)
  theta <- par[1:p]
  sigma <- par[p + 1]
  #constraint check
  #sigma (standard deviation) cannot be negative of zero
  #if optim tries a bad value, we return inf to push it back
  if (sigma <= 0) return(Inf)
  #calc loglikelihood
  ll <- LogLikelihood(theta, sigma, X, y)
  #calculate Ridge Penalty
  #Penalty = lambda * (sum of squared coefficients)
  #We do NOT penalize sigma, only the weights theta
  penalty <- lambda * sum(theta^2)
  #return minus Loglikelihood + penalty
  #we use minus because optim() searches for the MINIMUM
  #but we want to maximize likelihood
  return(-ll + penalty)
}

#==== C. RidgeOpt function ==
#Finds the best theta and sigma for a specific lambda
RidgeOpt <- function(lambda, X, y){
  p <- ncol(X)
  #initial guesses
  #we guess 0 for all weights, and 1 for standard deviation
  init_theta <- rep(0, p)
  init_sigma <- 1
  initial_params <- c(init_theta, init_sigma)
  #run optimization
  #optim() tries different values of "par" to make Ridge() as small as possible
  opt_results <- optim(par = initial_params,
                       fn = Ridge,
                       X = X,
                       y = y,
                       lambda = lambda,
                       method = "BFGS")
  #return the optimal parameters found
  return(opt_results$par)
}
#==== D. DF Function (Degrees of Freedom) ===
#calculates the effective nr of parameters.
DF <- function(lambda, X){
  #identity matrix
  I <- diag(ncol(X))
  #Compute X transpose X
  XtX <- t(X) %*% X
  #compute the matrix inverse part:
  inverse_part <- solve(XtX + lambda * I)
  #compute the Hat Matrix trace
  #Trace(AB) = Trace(BA)
  hat_trace <- sum(diag(inverse_part %*% XtX))
  return(hat_trace)
}

# --- Sanity Check for Task 2.3 Functions ---

# 1. Test LogLikelihood with dummy values
# We pretend theta is all zeros and sigma is 1
test_theta <- rep(0, ncol(X_train))
test_sigma <- 1
ll_val <- LogLikelihood(test_theta, test_sigma, X_train, y_train)
print(paste("LogLikelihood (Initial):", ll_val))
# Should be a negative number (e.g., -3000 to -5000)
# 2. Test Ridge Function with Lambda = 1
# We pass c(theta, sigma) as a single vector
test_params <- c(test_theta, test_sigma)
```

```
ridge_val <- Ridge(test_params, X_train, y_train, lambda = 1)
print(paste("Ridge Cost (Initial):", ridge_val))
# Should be slightly higher than -LogLikelihood (since penalty is added)
# 3. Test DF Function with Lambda = 1
df_val <- DF(lambda = 1, X = X_train)
print(paste("Degrees of Freedom (Lambda=1):", df_val))
# Should be close to the number of columns in your data (approx 16-17)

# ==== TASK 2.4 EVALUATION OR LAMDA = 1, 100 ,1000
#Define the lambdas to test
lambdas <- c(1, 100, 1000)
#loop through each lambda
for(lam in lambdas) {
  cat("\n=================\n")
  cat("ANALYZING LAMBDA =", lam, "\n")
  #A. Optimize (Find best theta and sigma)
  opt_params <- RidgeOpt(lambda = lam, X = X_train, y = y_train)
  #extract just the theta (weights)
  #the last element is sigma, so we remove it.
  p <- ncol(X_train)
  theta_hat <- opt_params[1:p]
  sigma_hat <- opt_params[p+1]
  #predict on Training and Test Data
  #prediction formula: y=X*theta
  pred_train <- X_train %*% theta_hat
  pred_test <- X_test %*% theta_hat
  #compute MSE
  mse_train <- mean((y_train - pred_train)^2)
  mse_test <- mean((y_test - pred_test)^2)
  #compute Degrees of Freedom
  df_val <- DF(lambda = lam, X = X_train)
  #results:
  cat("Training MSE:        ", mse_train, "\n")
  cat("Test MSE:            ", mse_test, "\n")
  cat("Degrees of Freedom: ", df_val, "\n")
}
```

1. **Pre-processing (Scaling Intercept):**

- **Why Scale?** Ridge penalizes the magnitude of coefficients ($||\theta||^2$). If features have different units (e.g., 0.001 vs 1000), the penalty will unfairly suppress the small-unit features. Scaling ensures λ affects all features equally.
- **Why No Intercept?** Since data is centered (mean=0) via scaling, the regression line passes through the origin $(0, 0)$. The intercept term becomes 0.

2. **Linear Regression (OLS) Issues:**

- **Multicollinearity:** Voice features (e.g., Jitter:Abs, Jitter:RAP) are highly correlated. This causes high variance in OLS estimates, making individual coefficients unstable and standard errors large (leading to misleading p-values regarding ”significance”).

3. **Ridge Regression Theory:**

- **Log-Likelihood (Normal):** For $y \sim \mathcal{N}(X\theta, \sigma^2 I)$:

$$LL(\theta, \sigma) \propto -\frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i^T\theta)^2$$

- **Ridge Cost Function:** We minimize the *negative* log-likelihood plus penalty:

$$J(\theta) = -LL + \lambda \sum \theta_j^2$$

Note: As $\lambda \to \infty$, the penalty dominates, forcing $\theta \to 0$ (Bias increases, Variance decreases).
- **Degrees of Freedom (DF):**

$$DF(\lambda) = tr(X(X^TX + \lambda I)^{-1}X^T)$$

- **Interpretation of DF:** Represents the "effective" number of parameters.
  - $\lambda = 0 \Rightarrow DF = p$ (Full OLS).
  - $\lambda \to \infty \Rightarrow DF \to 0$ (Null model).

4. Model Comparison ($\lambda = 1, 100, 1000$):

- $\lambda = 1$: Low penalty. High DF (close to feature count). Likely Overfitting (Low Train MSE, Higher Test MSE).
- $\lambda = 1000$: High penalty. Low DF. Likely Underfitting (High Train MSE, High Test MSE).
- **Selection:** The optimal λ (e.g., 100) balances Bias and Variance, achieving the lowest Test MSE.

Assignment 3: Logistic Regression & Basis Expansion

Problem Statement:
Data: *pima-indians-diabetes.csv.* ***Features:*** *Plasma Glucose (x_1), Age (x_2).*

1. *Load data, rename columns. Scatterplot of Glucose vs Age, colored by Diabetes.*
2. *Fit Logistic Regression: $Diabetes \sim Glucose + Age$. Report error ($r = 0.5$). Plot predictions.*
3. *Repeat classification/plotting for thresholds $r = 0.2$ and $r = 0.8$.*
4. ***Basis Function Expansion:*** *Add features $z_1 = x_1^4, z_2 = x_1^3 x_2, \ldots, z_5 = x_2^4$. Fit new model. Report error and plot.*

```r
library(ggplot2)
#===== data setup ===== 9 columns
data <- read.csv("pima-indians-diabetes.csv", header = FALSE)
#rename columns to match the assignment description
colnames(data) <- c("pregnant", "plasma_glucose", "distolic_bp",
                    "skinfold", "insulin", "BMI", "pedigree",
                    "age", "diabetes")
#verify data
#head(data)
#Important: convert diabetes to a "factor" so R knows it is a category (0 or 1)
#otherwise it might plot a color gradient
data$diabetes <- as.factor(data$diabetes)
#generate scatterplot
ggplot(data, aes(x=plasma_glucose, y=age, color = diabetes)) +
  geom_point() +
  labs(title = "Plasma Glucose vs Age (colored by diabetes)",
       x = "age (years)",
       y = "plasma glucose concentration")
  theme_minimal()

  #==== 3.2 logistic regression
  #train model with target diabetes, featured plasma glucose and age
  model_glm <- glm(diabetes ~ plasma_glucose + age,
                   family = "binomial",
                   data = data)
  #view coefficients for the equation
  summary(model_glm)
  #get probabilities
  probs <- predict(model_glm, type = "response")
  #classify based on r = 0.5
  preds <- ifelse(probs > 0.5, 1, 0)
  #compute missclassification error
  #only want 1 or 0
  actuals <- as.numeric(as.character(data$diabetes))
  error_rate <- mean(preds != actuals)
  print(error_rate)
  #add predictiions to dataframe for plotting
  data$Predicted <- as.factor(preds)
  #Plot
  ggplot(data, aes(x = plasma_glucose, y = age, color = Predicted)) +
    geom_point() +
    labs(title = "Plasma Glucose vs Age (Colored by PREDICTED Diabetes)",
         x = "Age",
         y = "Plasma Glucose") +
    theme_minimal()
  preds02 <- ifelse(probs > 0.2, 1, 0)
  preds08 <- ifelse(probs > 0.8, 1, 0)
  data$Predicted02 <- as.factor(preds02)
  data$Predicted08 <- as.factor(preds08)
  ggplot(data, aes(x = plasma_glucose, y = age, color = Predicted02)) +
    geom_point() +
    labs(title = "Plasma Glucose vs Age (Colored by PREDICTED Diabetes)",
         x = "Age",
         y = "Plasma Glucose") +
    theme_minimal()
  ggplot(data, aes(x = plasma_glucose, y = age, color = Predicted08)) +
    geom_point() +
    labs(title = "Plasma Glucose vs Age (Colored by PREDICTED Diabetes)",
         x = "Age",
         y = "Plasma Glucose") +
    theme_minimal()
  # 1. Create the new features
  # It is cleaner to define x1 and x2 variables first to match the formulas
  x1 <- data$plasma_glucose
  x2 <- data$age
  #just as the instructions
  data$z1 <- x1^4
  data$z2 <- (x1^3) * x2
  data$z3 <- (x1^2) * (x2^2)
  data$z4 <- x1 * (x2^3)
  data$z5 <- x2^4
  #train model with y as target as x1,x2,z1....z5 as features
  model_glm2 <- glm(diabetes ~ plasma_glucose + age + z1 +
                      z2+z3+z4+z5,
                    family = "binomial",
                    data = data)
  summary(model_glm2)
#predicrt probabilities new model
probs_glm2 <- predict(model_glm2, type = "response")
#threshold 0.5
preds_glm2 <- ifelse(probs_glm2 > 0.5, 1, 0)
#missclassification error
actuals <- as.numeric(as.character(data$diabetes)) #ensure numeric (0 or 1)
error_rate_glm2 <- mean(preds_glm2 != actuals)
print(error_rate_glm2)
#plot new predictions
data$predicted_glm2 <- as.factor(preds_glm2)
ggplot(data, aes(x = plasma_glucose, y = age, color = predicted_glm2)) +
  geom_point() +
  labs(title = "Predictions with Basis Function Expansion",
       y = "Age",
       x = "Plasma Glucose") +
  theme_minimal()
```

Interpretation:

- **Linear Boundary (Task 2):** Standard Logistic Regression creates a straight line. It fails to capture the complex, overlapping structure of the data, resulting in higher error.
- **Thresholds (Task 3):** $r = 0.2$ classifies almost everyone as diabetic (High Recall, Low Precision). $r = 0.8$ classifies very few (High Precision, Low Recall).
- **Basis Expansion (Task 4):** Adding polynomial terms (x^4, etc.) creates a non-linear (curved) decision boundary. This fits the data better (lower error) but increases model complexity.

Lab 2

Assignment 1: Explicit Regularization (LASSO & Ridge)

Problem Statement:
Data: *tecator.csv.* ***Target:*** *Fat.* ***Features:*** *Channels 1-100.*

0. ***Setup:*** *Import data. Split randomly into Train/Test (50/50) using standard lecture code.*
1. ***Linear Regression:*** *Model Fat using Channels. Report the underlying probabilistic model. Fit standard linear regression. Report Train/Test MSE. Comment on quality.*
2. ***LASSO Theory:*** *State the cost function for LASSO regression.*
3. ***LASSO Path:*** *Fit LASSO model. Plot regression coefficients vs* $\log \lambda$. *Interpret the plot. Determine the penalty factor (λ) that selects exactly* ***3 features****.*
4. ***Ridge Path:*** *Repeat step 3 but for Ridge Regression. Compare the coefficient plots of Ridge vs LASSO.*
5. ***CV LASSO:*** *Use Cross-Validation (CV) to find optimal LASSO model. Plot CV score vs* $\log \lambda$.
   - *Report optimal λ and number of selected variables.*
   - *Is the optimal model statistically significantly better than* $\log \lambda = -4$*?*
   - *Scatter plot: Original Test vs Predicted Test values for optimal model.*

```r
library(glmnet)
library(ggplot2)
# data setup
data <- read.csv("tecator.csv")
#split data
n <- nrow(data)
set.seed(12345)
id <- sample(1:n, floor(0.5 * n))
train_data <- data[id, ]
test_data <- data[-id, ]
#remove moisture and protein from dataset for training
train_clean <- subset(train_data, select = -c(Protein, Moisture))
test_clean <- subset(test_data, select = -c(Protein, Moisture))
print(dim(train_clean))
#fit linear regression
#fat ~ ,. means fat is predicted on all other columns in the dataframe
model_linear <- lm(Fat ~ ., data = train_clean)
summary(model_linear)
#predict on training data
preds_train <- predict(model_linear, newdata = train_clean)
mse_train <- mean((train_clean$Fat - preds_train)^2)
#predict on test data
preds_test <- predict(model_linear, newdata = test_clean)
mse_test <- mean((test_clean$Fat - preds_test)^2)
print("mse train: ")
print(mse_train)
print("mse test: ")
print(mse_test)

#==== assignment 1.2/1.3 Lasso
#glmnet only accepts matrices, not dataframes
#convert features to matrix and the target into a vector
#columns 2 to 101 as features (all the channels)
x_train <- as.matrix(train_clean[, 2:101])
#select fat as target (should be 102 when inspecting data)
y_train <- train_data$Fat
print(dim(x_train)) # Should be approx 50-100 rows x 100 columns
#Fit LASSO!!
#alpha = 1 (lasso), = 0 (ridge)
model_lasso <- glmnet(x_train, y_train, alpha = 1, family = "gaussian")
#Plot coefficient path
```

```
plot(model_lasso, xvar = "lambda", label = TRUE, main = "LASSO Coeff Path")
#find lambda (penalty) for exactly 3 features.
#find where df equals 3
print(model_lasso$df)
#extract lambda where df is 3
lambdas3 <- model_lasso$lambda[which(model_lasso$df == 3)]
print(lambdas3)
print(log(lambdas3))

#======== 1.4 RIDGE REGRESSION
#Fit LASSO!!
#alpha = 1 (lasso), = 0 (ridge)
model_ridge <- glmnet(x_train, y_train, alpha = 0, family = "gaussian")
#Plot coefficient path
plot(model_ridge, xvar = "lambda", label = TRUE, main = "RIDGE Coeff Path")

#1.5 Cross Validation =================
set.seed(12345)
#run CV (cross validation)
cv_lasso <- cv.glmnet(x_train, y_train, alpha = 1, family = "gaussian")
plot(cv_lasso)
#optimal lambda
best_lambda <- cv_lasso$lambda.min
print(best_lambda)
print(log(best_lambda))
#check how many variables are chosen
coef_optimal <- coef(cv_lasso, s = "lambda.min")
#count non zero coefficients (-1 for intercept)
num_vars <- sum(coef_optimal != 0) -1
print(num_vars)
#==== final scatter plot, verify model quality on test data
#prepare data just like x_train
x_test <- as.matrix(test_data[, 2:101])
y_test <- as.matrix(test_data$Fat)
preds <- predict(cv_lasso, newx = x_test, s = "lambda.min")
#create scatter plot (predicted vs actual)
df_plot <- data.frame(Actual = y_test, Predicted = as.vector(preds))
ggplot(df_plot, aes(x = Actual, y = Predicted)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, color = "red") + # Perfect prediction line
  labs(title = "LASSO: Predicted vs Actual Fat Content",
       x = "Actual Fat",
       y = "Predicted Fat") +
  theme_minimal()
```

Theory Answers:

- **1. Probabilistic Model:** $y \mid x \sim N(\theta^T x, \sigma^2)$. The errors are normally distributed.
- **3. LASSO Cost: w**$) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 + \lambda\sum_{j=1}^{p}|w_j|$. The last term is known as the L1 norm of the weight vector. The Wj notation = absolute value of each weight. By penalizing the sum of these absolute values, the cost function encourages the model to be sparse. This means that during optimization, many of the coefficients Wj will be driven to exactly zero, removing those features from the model.
- **4 vs 5. Ridge vs LASSO:** LASSO coefficients hit exactly zero as λ increases (Variable Selection). Ridge coefficients shrink towards zero but never truly reach it (Smooth shrinkage).
- **6. Significance:** If the CV score of log $\lambda = -4$ is outside the error bars of the optimal model, the optimal is significantly better. If it is within the bars, they are statistically comparable.

Assignment 2: Decision Trees & Logistic Regression (Bank Marketing)

Problem Statement:
Data: *bank-full.csv.* ***Target:*** *y (Term deposit subscription: 'yes'/'no').*

1. ***Setup:*** *Import data. Remove variable `duration` (it leaks target info). Split into Training/Validation/Test (40/30/30).*
2. ***Tree Models:*** *Fit three decision trees to Training data:*
   (a) *Default settings.*
   (b) *Smallest node size = 7000.*
   (c) *Minimum deviance = 0.0005.*

   Report Train/Validation misclassification rates. Choose the best model. Explain how deviance and node size affect tree size.
3. ***Optimal Tree Depth:*** *For model 2c (min deviance 0.0005), study trees up to 50 leaves.*
   - *Plot Deviance (Train & Validation) vs Number of Leaves. Interpret in terms of Bias-Variance.*
   - *Report optimal number of leaves and most important variables.*
   - *Interpret the tree structure (key findings).*
4. ***Evaluation:*** *Report Confusion Matrix, Accuracy, and F1 Score on Test data using the optimal model. Comment on predictive power and which metric (Accuracy vs F1) is preferred.*
5. ***Loss Matrix:*** *Perform classification on Test data using a specific Loss Matrix:*
   - *Loss(Obs='yes', Pred='no') = 1*
   - *Loss(Obs='no', Pred='yes') = 5*

   Report Confusion Matrix. Compare with Step 4 results.
6. ***ROC & Logistic Regression:*** *Train a Logistic Regression model. Use both the Optimal Tree and Logistic Regression to classify Test data with probability thresholds* $\pi = 0.05, 0.1, \ldots, 0.95$.
   - *Plot ROC curves (TPR vs FPR) for both models.*
   - *Discuss: Why might a Precision-Recall curve be a better option here?*

```r
# SubTask 1. ========================
# Read bank-full and remove duration
myData <- read.csv2("bank-full.csv", stringsAsFactors = TRUE)
myData <-subset(myData, select=-duration)
# Partition myData into train, valid, and test
n <- dim(myData)[1]
set.seed(12345)
id <- sample(1:n, floor(n*0.4))
id1 <- setdiff(1:n, id)
set.seed(12345)
id2 <- sample(id1, floor(n*0.3))
id3 <- setdiff(id1,id2)
train <- myData[id,]
valid <- myData[id2,]
test <- myData[id3,]

# SubTask 2. ============================
library(tree)
n=dim(myData)[1]
# a.
treeDefault <- tree(y~., data=train)
plot(treeDefault)
text(treeDefault, pretty=0)
treeDefault
summary(treeDefault)
Ydefault <- predict(treeDefault, newdata=train, type='class')
table(train$y, Ydefault)
# b.
treeLarge <- tree(y~., data=train, control=tree.control(nobs=nrow(train),minsize=7000))
plot(treeLarge)
```

```r
text(treeLarge, pretty=0)
treeLarge
summary(treeLarge)
Ylarge <- predict(treeLarge, newdata=train, type='class')
table(train$y, Ylarge)
# c.
treeSmall <- tree(y~., data=train, control=tree.control(nobs=nrow(train), mindev=0.0005))
plot(treeSmall)
text(treeSmall, pretty=1)
treeSmall
summary(treeSmall)
Ysmall <- predict(treeSmall, newdata=train, type='class')
table(train$y, Ysmall)
misClassError <- function(tree, newdata){
  pred <- predict(tree, newdata=newdata, type='class')
  mean(pred != newdata$y)
}
mis_train_default <- misClassError(treeDefault, train)
mis_valid_default <- misClassError(treeDefault, valid)
mis_train_large <- misClassError(treeLarge, train)
mis_valid_large <- misClassError(treeLarge, valid)
mis_train_small <- misClassError(treeSmall, train)
mis_valid_small <- misClassError(treeSmall, valid)

# Subtask 3. ======================================
trainScore <- rep(0,50)
validScore <- rep(0,50)
for (i in 2:50) {
  treePruned <- prune.tree(treeSmall, best=i)
  pred <- predict(treePruned, newdata=valid, type='tree')
  trainScore[i] <- deviance(treePruned)
  validScore[i] <- deviance(pred)
}
plot(2:50, trainScore[2:50], type='b', col='red', ylim = c(8200,12000))
points(2:50, validScore[2:50], type='b', col='blue')
legend("topright",
       legend=c("Training deviance", "Validation deviance"),
       col=c("red", "blue"), lty=1, pch=19)
treeOpt <- prune.tree(treeSmall, best=22)
plot(treeOpt)
text(treeOpt, pretty=0)

# Subtask 4. =====================================
# Predict
predTest <- predict(treeOpt, newdata=test, type='class')
# Create confusion matrix
confMat <- table(Actual = test$y, Predicted = predTest)
confMat
# Calculate accuracy
accuracy <- sum(diag(confMat)/sum(confMat))
accuracy
# Find F1 score
# Extract True positive, False positive and False Negative
TP <- confMat["yes", "yes"]
FP <- confMat["no", "yes"]
FN <- confMat["yes", "no"]
precision <- TP / (TP+FP)
recall <- TP / (TP+FN)
F1 <- 2 * precision * recall / (precision+recall)
F1

# Subtask 5. =====================================
# Create loss matrix
lossMatrix <- matrix(c(
  0, 1,
  5, 0
), nrow=2, byrow=TRUE)
rownames(lossMatrix) <- colnames(lossMatrix) <- c("no", "yes")
probs <- predict(treeOpt, test, type="vector")
p_yes <- probs[, "yes"]
threshold <-1/6
pred_thresh <- ifelse(p_yes > threshold, "yes", "no")
pred_thresh <- factor(pred_thresh, levels=c("no","yes"))
table(Actual=test$y, Predicted=pred_thresh)

# Subtask 6. ========================================
logModel <- glm(y~., data=train, family=binomial)
tree_probs <- predict(treeOpt, test, type="vector")[, "yes"]
log_probs <- predict(logModel, test, type="response")
thresholds <- seq(0.05, 0.95, by=0.05)
getTRPFPR <- function(probs, true_y, thresh){
  pred <- ifelse(probs > thresh, "yes", "no")
  pred <- factor(pred, levels=c("no", "yes"))
  cm <- table(Predicted=pred, Actual=true_y)
  TP <- cm["yes","yes"]
  FP <- cm["yes", "no"]
  FN <- cm["no", "yes"]
  TN <- cm["no", "no"]
  TPR <- TP / (TP+FN)
  FPR <- FP / (FP+TN)
  c(TPR = TPR, FPR = FPR)
}
tree_roc <- t(sapply(thresholds, function(t){
  getTRPFPR(tree_probs,test$y,t)
}))
log_roc <- t(sapply(thresholds, function(t){
  getTRPFPR(log_probs, test$y, t)
}))
plot(tree_roc[,"FPR"], tree_roc[,"TPR"], type="b",
     xlab="False Positive Rate (FPR)",
     ylab="True Positive Rate (TPR)",
     main="ROC Curves: Tree vs Logistic Regression",
     pch=19)
lines(log_roc[,"FPR"], log_roc[,"TPR"], type="b", col="red", pch=19)
legend("bottomright",
       legend=c("Decision Tree", "Logistic Regression"),
       col=c("black", "red"), lty=1, pch=19)
tree_roc
log_roc
```

Interpretation:

- **2. Tree Models:**
  - **Smallest node size (7000):** Forces the tree to be very small/shallow (High Bias, Underfitting).
  - **Min deviance (0.0005):** Allows the tree to grow very deep/complex (High Variance, Overfitting).
- **3. Deviance Plot (Bias-Variance):** Training deviance (Red) decreases continuously. Validation deviance (Blue) decreases initially but then rises (U-shape). The minimum of the Validation curve indicates the optimal depth (balancing bias and variance).
- **4. Accuracy vs F1:** The dataset is imbalanced (mostly 'no'). A model predicting 'no' for everyone would have high Accuracy but 0 F1. **F1 Score** is preferred as it measures the ability to detect the minority class ('yes').
- **5. Loss Matrix:** Penalizing False Negatives (5x cost) lowers the decision threshold ($1/(1+5) \approx 0.16$). This increases Recall (more 'yes' predicted) but lowers Precision (more false alarms).
- **6. ROC vs Precision-Recall:** ROC curves can be misleading on imbalanced data because the False Positive Rate (FP/N) stays low when N is huge. Precision-Recall curves focus on the minority class and are often more

informative here.

Assignment 3: PCA & Implicit Regularization (Crime Data)

Problem Statement:
Data: *communities.csv.* ***Target:*** *ViolentCrimesPerPop.*

1. ***PCA (Eigen):*** *Scale features (exclude target). Implement PCA using eigen().*
   - *Report how many components are needed to obtain $\geq 95\%$ variance.*
   - *Report proportion of variance explained by PC1 and PC2.*
2. ***PCA (Princomp) & Analysis:*** *Repeat PCA using princomp().*
   - ***Trace Plot:*** *Plot loadings of PC1. Do many features contribute?*
   - ***Top 5 Features:*** *Report the 5 features with largest absolute weights in PC1. Discuss their logical relationship to crime.*
   - ***Score Plot:*** *Plot PC1 vs PC2, colored by ViolentCrimesPerPop. Analyze the pattern.*
3. ***Linear Regression:*** *Split 50/50. Scale both features and target. Fit Linear Regression. Report Train/Test MSE.*
4. ***Implicit Regularization (Early Stopping):***
   - *Implement the Linear Regression cost function (MSE without intercept).*
   - *Optimize using optim(method="BFGS") starting from $\theta = 0$, so without gradient specified.*
   - ***Track Progress:*** *Compute Train/Test MSE for every iteration.*
   - ***Plot:*** *Error vs Iteration (discard first 500 iters for visibility).*
   - ***Conclusion:*** *Find the optimal iteration (Early Stopping). Compare Test MSE with Step 3.*

```r
#Data prep
data <- read.csv("communities.csv", stringsAsFactors =TRUE)
#separate Target and Features
#scale features but NOT the target ViolentCrimesPerPop
features <- data[, -which(names(data) == "ViolentCrimesPerPop")]
target <- data$ViolentCrimesPerPop
#scale the features!
features_scaled <- scale(features)
#implement PCA with eigen()
#calc the covariance matrix
cov_matrix <- cov(features_scaled)
#calculate eigenvalues and eigenvectors
#eigen() returns a list of containign $values (Eigenvalues) and $vectors
pca_eigen <- eigen(cov_matrix)
eigenvalues <- pca_eigen$values
#calculate Variance explained
#proportion of variance = eigenvalue / sum of all eigenvalues
var_explained <- eigenvalues / sum(eigenvalues)
cum_var_explained <- cumsum(var_explained) #cumsum = cumulative sum
#report results
#how many components for at least 95% variance?
num_components_95 <- which(cum_var_explained >= 0.95)[1]
#Proportion of variation explained by the first two components
prop_pc1 <- var_explained[1]
prop_pc2 <- var_explained[2]
cat("Number of components needed for 95% variance:", num_components_95, "\n")
cat("Proportion of variance explained by PC1:", round(prop_pc1 * 100, 2), "%\n")
cat("Proportion of variance explained by PC2:", round(prop_pc2 * 100, 2), "%\n")

#=== Task 2 =====================
library(ggplot2)
#run PCA using princomp()
#we use the scaled features from Step 1
pca_res <- princomp(features_scaled)
```

```r
#trace plot fo the first principal component
#shows the loadings (weights) of every variable in PC1
U1 <- pca_res$loadings[, 1]
#plot loadings
plot(U1, main = "Trace Plot of PC1 Loadings",
     xlab = "Variable Index", ylab = "Loading Value", type = "h")
# Add a horizontal line at 0 for reference
abline(h = 0, col = "red")
#report top 5 contributing features
#we look at the absolute value because a large negatvie weight is as strong as a pos
top_5_indices <- order(abs(U1), decreasing = TRUE)[1:5]
top_5_features <- U1[top_5_indices]
cat("Top 5 features contributing to PC1:\n")
print(top_5_features)
# 4. Score Plot (PC1 vs PC2) colored by ViolentCrimesPerPop
# Create a data frame for plotting
df_pca <- data.frame(
  PC1 = pca_res$scores[, 1],
  PC2 = pca_res$scores[, 2],
  Crime = target
)
# Use ggplot for the colored scatter plot
ggplot(df_pca, aes(x = PC1, y = PC2, color = Crime)) +
  geom_point(alpha = 0.6) +
  scale_color_gradient(low = "blue", high = "red") +
  labs(title = "PCA Score Plot: Crime Level",
       x = "First Principal Component (PC1)",
       y = "Second Principal Component (PC2)") +
  theme_minimal()

#==== TASK 3===
#Prepare data and split 50 50
set.seed(12345)
n <- nrow(data)
id <- sample(1:n, size = n)
train_indices <- id[1:floor(n/2)]
test_indices <- id[(floor(n/2) + 1):n]
train_data <- data[train_indices, ]
test_data <- data[test_indices, ]
#scale the data, scale() returns a matrix, so we use as.data.frame
train_scaled <- as.data.frame(scale(train_data))
#OBS: We scale the TEST data using the MEAN and SD of the TRAINING data
#ensures we dont cheat by using information from test set
train_means <- attr(scale(train_data), "scaled:center")
train_sds <- attr(scale(train_data), "scaled:scale") #sd = standard dev
#apply these to test data
test_scaled <- as.data.frame(scale(test_data, center = train_means, scale = train_sds))
#fit lin reg model
#target: ViolentCrimesPerPop, Features: All other (.)
m1 <- lm(ViolentCrimesPerPop ~ ., data = train_scaled)
#compute MSE
#predict on training data
pred_train <- predict(m1, newdata = train_scaled)
mse_train <- mean((train_scaled$ViolentCrimesPerPop - pred_train)^2)
#prediction on test data
pred_test <- predict(m1, newdata = test_scaled)
mse_test <- mean((test_scaled$ViolentCrimesPerPop - pred_test)^2)
# 5. Report Results
cat("Linear Regression Results:\n")
cat("Training MSE:", mse_train, "\n")
cat("Test MSE:    ", mse_test, "\n")
# Optional: R-squared to check fit quality
```

```r
cat("Training R-squared:", summary(m1)$r.squared, "\n")

#==== TASK 4=======
#prepare matrices
#use matrices bcs we need to do matrix multiplication (X * theta)
#the assignment specifices "Linear Regression WITHOUT intercept)
#so we just use the scaled data directly
train_x_mat <- as.matrix(train_scaled[, -which(names(train_scaled) == "ViolentCrimesPerPop")])
train_y_mat <- as.matrix(train_scaled$ViolentCrimesPerPop)
test_x_mat <- as.matrix(test_scaled[, -which(names(test_scaled) == "ViolentCrimesPerPop")])
test_y_mat <- as.matrix(test_scaled$ViolentCrimesPerPop)
#initialize global variables to store the history
#we ned these to exist outside the function so we can write to them
iteration_log <- data.frame(Train_MSE = numeric(), Test_MSE = numeric())
#Cost function (MSE)
#this function calculates the error for a given "theta"
#it also has the side effect of saving the errors to our log
cost_function <- function(theta, X_train, y_train, X_test, y_test){
  #calculate prediction for training
  #forumla: y_pred = X * theta
  pred_train <- X_train %*% theta
  #calc MSE (want to minimize)
  mse_train <- mean((y_train - pred_train)^2)
  #side effect tack test error
  #we calc how this theta performs on the test set right now
  pred_test <- X_test %*% theta
  mse_test <- mean((y_test - pred_test)^2)
  #apend to the global log (using the <<- operator)
  iteration_log <<- rbind(iteration_log,data.frame(Train_MSE = mse_train, Test_MSE = mse_test))
  #return the value to be minimzed (training error)
  return(mse_train)
}
#Run Optimization (BFGS)
#initial theta is all zeros (per instructions)
theta_start <- rep(0, ncol(train_x_mat))
#reset log before running
iteration_log <- data.frame(Train_MSE = numeric(), Test_MSE = numeric())
#run optim()
#method="BFGS" uses gradients, but will approximate them numerically since we have no gr
opt_res <- optim(par=theta_start, fn = cost_function, method = "BFGS",
                 X_train = train_x_mat, y_train = train_y_mat,
                 X_test = test_x_mat, y_test = test_y_mat)
#process the results
#the assignment asks to discard the first 500 iteration to make the plot readable
plot_data <- iteration_log[501:nrow(iteration_log), ]
plot_data$Iteration <- 501:nrow(iteration_log) #create index
#plot training vs test error
library(ggplot2)
ggplot(plot_data, aes(x = Iteration)) +
  geom_line(aes(y = Train_MSE, color = "Training Error")) +
  geom_line(aes(y = Test_MSE, color = "Test Error")) +
  coord_cartesian(ylim = c(0 , 0.6)) +
  scale_color_manual(values = c("Training Error" = "black", "Test Error" = "red")) +
  labs(title = "Optimization History (BFGS)", y = "Mean Squared Error", x = "Iteration Number") +
  theme_minimal()
# 7. Find Optimal Iteration (Early Stopping)
# We want the iteration where Test Error was at its Minimum
min_test_index <- which.min(iteration_log$Test_MSE)
optimal_test_error <- iteration_log$Test_MSE[min_test_index]
optimal_train_error <- iteration_log$Train_MSE[min_test_index]
cat("Optimal Iteration Number:", min_test_index, "\n")
cat("Minimum Test MSE (Early Stopping):", optimal_test_error, "\n")
cat("Training MSE at that point:", optimal_train_error, "\n")
```

Interpretation:

- **PCA:** Many variables contribute slightly to PC1. Features related to family structure (e.g., `medFamInc`, `pctWInvInc`) often dominate. Crime correlates with PC1/PC2 clusters.
- **Implicit Regularization:** BFGS optimization starting from 0 acts like regularization. Early iterations correspond to high λ (high bias, low variance). As iterations increase, the model becomes more complex. Stopping early (at the minimum Test MSE) prevents overfitting, similar to choosing an optimal Ridge penalty.

Lab 3

Assignment 2: Kernel Methods (Temperature Prediction)

Problem Statement:
Data: *stations.csv (coords), temps50k.csv (measurements).* ***Goal:*** *Implement a Kernel Estimator to predict temperature at a specific location/date/time.*

1. ***Model:*** $\hat{y} = \frac{\sum K_i y_i}{\sum K_i}$.
2. ***Kernels:*** *Define three Gaussian kernels based on distance:*
   - ***Physical Distance (***h_{dist}***):*** *Great-circle dist (Haversine) between target and station.*
   - ***Day Distance (***h_{date}***):*** *Day of year diff (handling 365 $\to$ 1 wrap-around).*
   - ***Time Distance (***h_{time}***):*** *Hour of day diff (handling 24 $\to$ 0 wrap-around).*
3. ***Combined Kernel:*** *Product ($K_{sum} = K_{dist} + K_{date} + K_{time}$) OR Sum ($K_{prod} = K_{dist} \cdot K_{date} \cdot K_{time}$). *Note: Lab usually asks for Sum or Product logic, check specific instruction. Usually Product for "intersection" of conditions.**
4. ***Filter:*** *Prevent "predicting the future" by filtering out measurements after the target date.*
5. ***Task:*** *Predict & Plot temps for* ***Linköping*** *(58.4274, 15.662) for a range of times.*

```
#data setup
set.seed(1234567890)
library(geosphere)
stations <- read.csv("stations.csv", fileEncoding = "latin1")
temps <- read.csv("temps50k.csv")
st <- merge(stations, temps, by ="station_number")
#deifne smoothing coefficients (width) - tuned manually
h_distance <- 50000 #100km (in meters ty distHaversine)
h_date <- 20 #30 days
h_time <- 2 #4 hours
#target coordinates and date
a <- 58.0340 #latidude
b <- 14.9756 #longitude
pos_pred <- c(b, a) #distHaversine format: c(longitude, latitude)
date_pred <- "2013-08-16"
times <- c("04:00:00", "06:00:00", "08:00:00", "10:00:00", "12:00:00",
           "14:00:00", "16:00:00", "18:00:00", "20:00:00", "22:00:00", "24:00:00")
#pre calc distances for speed
#physical distance
#st$longitude and st$latitude are the columns
st_coords <- cbind(st$longitude, st$latitude)
st$dist_km <- distHaversine(st_coords, pos_pred)
#date distance
#convert to simple data objects to find the difference in days
st$date_obj <- as.Date(st$date)
target_date_obj <- as.Date(date_pred)
#distance in absolute difference in days
st$day_diff <- as.numeric(abs(st$date_obj - target_date_obj))
#time distance
#0-24 cycle
```

```
st$hour <- as.numeric(substr(st$time, 1, 2))
#prediction loop
preds <- numeric(length(times))
kernel_plots <- list() #storage for plotting kernel decays later
cat("Calculating predictions..\n")
for (i in 1:length(times)){
  target_time_str <- times[i]
  #handle 24:00:00 case
  target_hour <- as.numeric(substr(target_time_str, 1, 2))
  #filter out measurements posteriro to the moment of prediction
  #create temporary comparison column
  current_target_datetime <- as.POSIXct(paste(date_pred, target_time_str), format="%Y-%m-%d %H:%M
    ↪ :%S")
  valid_idx <- (st$date_obj < target_date_obj) | (st$date_obj == target_date_obj & st$hour <
    ↪ target_hour)
  st_filtered <- st[valid_idx, ]

  #---- KERNEL CALCULATION (on filtered data)
  #distance kernel
  dist_diff <- st_filtered$dist_km
  k_dist <- exp(-(dist_diff^2) / (2 * h_distance^2))
  #date kernel
  day_diff <- st_filtered$day_diff
  k_date <- exp(-(day_diff^2) / (2*h_date^2))
  #time kernel
  #calc hour diff
  hour_diff <- abs(st_filtered$hour - target_hour)
  k_time <- exp(-(hour_diff^2) / (2 * h_time^2))
  #COMBINE KERNELS (SUM)
  #sum of three gaussian kernels
  k_total <- k_dist + k_date + k_time
  #predict
  #weighted mean
  preds[i] <- sum(k_total * st_filtered$air_temperature) / sum(k_total)
}
#Plotting results
plot(1:length(times), preds, type="o", col="red", lwd=2, xaxt="n",
     xlab="Time", ylab="Predicted Temperature", main="Temp Forecast (Sum of Kernels)")
axis(1, at=1:length(times), labels=times)
print(data.frame(Time=times, Predicted_Temp=preds))
# 7. Plotting Kernel Values (As requested)
# We plot how the kernel weight decays as we move away from the target
par(mfrow=c(1,3))
# Distance Decay
d_seq <- seq(0, 300000, by=1000) # 0 to 300km
k_d_plot <- exp(-(d_seq^2)/(2*h_distance^2))
plot(d_seq/1000, k_d_plot, type="l", main="Distance Kernel", xlab="Distance (km)", ylab="Weight")
abline(v=h_distance/1000, col="red", lty=2) # Mark h
# Day Decay
day_seq <- seq(0, 60, by=1) # 0 to 60 days
k_day_plot <- exp(-(day_seq^2)/(2*h_date^2))
plot(day_seq, k_day_plot, type="l", main="Date Kernel", xlab="Difference (Days)", ylab="Weight")
abline(v=h_date, col="red", lty=2)
# Hour Decay
h_seq <- seq(0, 12, by=0.1) # 0 to 12 hours
k_h_plot <- exp(-(h_seq^2)/(2*h_time^2))
plot(h_seq, k_h_plot, type="l", main="Hour Kernel", xlab="Difference (Hours)", ylab="Weight")
abline(v=h_time, col="red", lty=2)
par(mfrow=c(1,1)) # Reset

#PRODUCT  kernels (oral defense part)
# --- 1. Initialize storage for Product predictions ---
```

```
preds_prod <- numeric(length(times))
cat("Calculating Product predictions...\n")
# --- 2. Loop for Product Predictions ---
for (i in 1:length(times)) {
  target_time_str <- times[i]
  target_hour <- as.numeric(substr(target_time_str, 1, 2))
  # Filter (same logic as before)
  # Only keep past data
  valid_idx <- (st$date_obj < target_date_obj) |
    (st$date_obj == target_date_obj & st$hour < target_hour)
  st_filtered <- st[valid_idx, ]
  # Recalculate Kernels for filtered data
  dist_diff <- st_filtered$dist_km
  k_dist <- exp(-(dist_diff^2) / (2 * h_distance^2))
  day_diff <- st_filtered$day_diff
  k_date <- exp(-(day_diff^2) / (2 * h_date^2))
  hour_diff <- abs(st_filtered$hour - target_hour)
  k_time <- exp(-(hour_diff^2) / (2 * h_time^2))
  # --- CRITICAL CHANGE: MULTIPLICATION ---
  # Product of kernels acts like "AND" logic
  k_total_prod <- k_dist * k_date * k_time
  # Predict
  preds_prod[i] <- sum(k_total_prod * st_filtered$air_temperature) / sum(k_total_prod)
}
# --- 3. Plot Comparison ---
# Plot Sum (Red) vs Product (Blue)
plot(1:length(times), preds, type="o", col="red", lwd=2, xaxt="n", ylim=c(min(preds, preds_prod)
    ↪ -2, max(preds, preds_prod)+2),
     xlab="Time", ylab="Predicted Temperature", main="Comparison: Sum vs. Product Kernel")
axis(1, at=1:length(times), labels=times)
lines(1:length(times), preds_prod, type="o", col="blue", lwd=2, lty=2)
legend("topleft", legend=c("Sum (OR logic)", "Product (AND logic)"),
       col=c("red", "blue"), lty=c(1, 2), lwd=2)
```

Interpretation (Sum vs. Product):

- **Sum (OR):** Includes data points that are close in distance OR close in time OR close in date. This uses a massive amount of data (global smoothing), resulting in a very smooth but ”flat” prediction that might miss local extremes (like the daily temp cycle).
- **Product (AND):** Includes only points that are close in distance AND time AND date. This isolates the specific context (local smoothing), capturing the daily temperature variation much better, but can be noisy if data is sparse.

Assignment 3: Support Vector Machines (SVM)

Problem Statement:
Data: *spam (from kernlab).* ***Target:*** *type (spam/nonspam).*

1. ***Basic SVM:*** *Train ksvm using RBF kernel (rbfdot) with default settings. Report Error.*
2. ***Tuning:*** *Train SVMs with fixed width sigma=0.05 (kpar=list(sigma=0.05)) and varying cost $C \in \{0.5, 1, 5\}$.*
3. ***Selection:*** *Select the best C based on generalization error (Cross-Validation or Hold-out).*
4. ***Theory:*** *Explain the role of C (Regularization parameter: Trade-off between margin width and training error).*

```
# Lab 3 block 1 of 732A99/TDDE01/732A68 Machine Learning
library(kernlab)
set.seed(1234567890)
data(spam)
foo <- sample(nrow(spam))
spam <- spam[foo,]
tr <- spam[1:3000, ]
va <- spam[3001:3800, ]
```

```r
trva <- spam[1:3800, ]
te <- spam[3801:4601, ]
by <- 0.3
err_va <- NULL
for(i in seq(by,5,by)){
  filter <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=i,scaled=FALSE)
  mailtype <- predict(filter,va[,-58])
  t <- table(mailtype,va[,58])
  err_va <-c(err_va,(t[1,2]+t[2,1])/sum(t))
}
filter0 <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=which.min(err_va)*by,
    ↪ scaled=FALSE)
mailtype <- predict(filter0,va[,-58])
t <- table(mailtype,va[,58])
err0 <- (t[1,2]+t[2,1])/sum(t)
err0
filter1 <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=which.min(err_va)*by,
    ↪ scaled=FALSE)
mailtype <- predict(filter1,te[,-58])
t <- table(mailtype,te[,58])
err1 <- (t[1,2]+t[2,1])/sum(t)
err1
filter2 <- ksvm(type~.,data=trva,kernel="rbfdot",kpar=list(sigma=0.05),C=which.min(err_va)*by,
    ↪ scaled=FALSE)
mailtype <- predict(filter2,te[,-58])
t <- table(mailtype,te[,58])
err2 <- (t[1,2]+t[2,1])/sum(t)
err2
filter3 <- ksvm(type~.,data=spam,kernel="rbfdot",kpar=list(sigma=0.05),C=which.min(err_va)*by,
    ↪ scaled=FALSE)
mailtype <- predict(filter3,te[,-58])
t <- table(mailtype,te[,58])
err3 <- (t[1,2]+t[2,1])/sum(t)
err3
# Questions
# 1. Which filter do we return to the user ? filter0, filter1, filter2 or filter3? Why?
#Filter 3 since it uses the full dataset (spam, 4601 rows) making it the most robust model
# 2. What is the estimate of the generalization error of the filter returned to the user? err0,
    ↪ err1, err2 or err3? Why?
#err2 is the best unbiased estimate, measures performance on the independent test et (te) using
#a model (filter2) trained on tr + va. This mimincs final scenario while still maintaining a
#strict eparation between training and testing data.

# 3. Implementation of SVM predictions.
sv<-alphaindex(filter3)[[1]] #indices of support vectors
co<-coef(filter3)[[1]] #Linear coefficients (alpha * y)
inte<- - b(filter3) #intercept (bias)
sigma <- 0.05 #defined in the problem
k<-NULL
for(i in 1:10){ # We produce predictions for just the first 10 points in the dataset.
  k2<-0 #intit the sum for this point
  for(j in 1:length(sv)){ #loop over all support vectors
    #get the support vector data point (excluding target column 58)
    support_vector <- spam[sv[j], -58]
    #get the point we are predicting
    x_point <- spam[i, -58]
    #calculate Squared Euclidean Distance
    #we convert to numeric vectors to ensure math operations work
    dist_sq <- sum((as.numeric(support_vector) - as.numeric(x_point))^2)
    #calculate RBF Kernel value
    kernel_val <- exp(-sigma * dist_sq)
    #add weighted contribution to the sum
    k2 <- k2 + (co[j] * kernel_val)
  }
  #add the intercept and store the final decision vaue
  k<-c(k, k2 + inte) # Your code here)
}
#verification
cat("Manual Calc:\n")
print(k)
cat("\nBuilt in predict function:\n")
print(predict(filter3,spam[1:10,-58], type = "decision"))
```

Theory Answers:

- **Which Filter? (Q1):** We return filter3. Once hyperparameters are tuned and performance is verified, we should retrain on **all available data** (Train + Valid + Test) to maximize the information the model learns from.
- **Generalization Error? (Q2):** We report err2 (or err1). err2 is the error of a model trained on (Train+Valid) and evaluated on (Test). This is the closest proxy to how the final model (filter3) will perform on unseen data, as the Test set was never used for training or tuning.

Assignment 4: Neural Networks (Sine Regression)

Problem Statement:
Data: *Generated synthetic data.*

1. ***Forward Task (*** $x \to \sin(x)$ ***):***
   - *Generate 50 points* $x \sim U[0, 10]$, $y = \sin(x)$.
   - *Train NN to predict* y *from* x.
   - *Plot predictions on Training data (Black dots) vs NN (Red line).*
2. ***Generalization:*** *Generate 500 new points* $x \sim U[0, 10]$. *Predict and Plot.*
3. ***Inverse Task (*** $\sin(x) \to x$ ***):***
   - *Train NN to predict* x *given* $\sin(x)$ *using 500 training points.*
   - *Predict on Training data. Plot results.*
   - ***Result:*** *It will fail (converge to average) because the inverse of Sine is not a function (one-to-many mapping).*

```
library(neuralnet)
set.seed(1234567890)
#data generation
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin = sin(Var))
#split data
tr <- mydata[1:25, ] #training 25 points
te <- mydata[25:500, ] #test 475
#random init of weights
#instructions asks for "winit". For a 1-10-1 network, theres:
#input -> hidden: (1 input + 1 bias) * 10 neurons = 20 weights
#Hidden -> output: (10 hidden + 1 bias) * 1 neuron = 11 weights
#Total 31 weights, init them randomly in [-1, 1]
winit <- runif(31, -1, 1)
#Train NN
nn <- neuralnet(Sin ~ Var, data = tr, hidden = 10, startweights = winit)
#results
plot(tr, col = "black", cex = 2,
     xlim = c(0,10), ylim = c(-1.5, 1.5),
     main = "NN Regression: Sin(x)", xlab = "Var", ylab = "Sin")
#plot test data
points(te, col = "blue", cex = 1, pch=19)
#Plot predictions on test data (red)
preds <- predict(nn, te)
points(te[,1], preds, col = "red", cex = 1, pch = 3)
```

```
legend("bottomleft", legend = c("Train", "Test", "Predicted"),
       col = c("black", "blue", "red"), pch=c(19, 1, 3))

#==== 4.2 Custom activation
linear_act <- function(x) { x }
relu_act <- function(x) { ifelse(x > 0, x, 0) }
softplus_act <- function(x) { log(1 + exp(x)) }
#setup plottin area (1 row 3 columns)
par(mfrow = c(1, 3))
#model A linear --------
set.seed(1234567890)
winit <- runif(31, -1, 1) #reset weights for fairness
nn_lin <- neuralnet(Sin ~ Var, data = tr, hidden = 10, startweights = winit,
                    act.fct = linear_act)
#plot linear
plot(tr, col = "black", main = "Linear Activation", ylim = c(-1.5, 1.5), pch = 19)
points(te, col = "blue", cex = 0.5)
points(te[,1], predict(nn_lin, te), col = "red", cex = 1)
#model B RELUU-----------------
set.seed(1234567890)
winit <- runif(31, -1, 1)
nn_relu <- neuralnet(Sin ~ Var, data = tr, hidden = 10, startweights = winit,
                     act.fct = relu_act)
#plot
plot(tr, col = "black", main = "RelU Activation", ylim = c(-1.5, 1.5), pch = 19)
points(te, col = "blue", cex = 0.5)
points(te[,1], predict(nn_relu, te), col = "red", cex=1)
#model C SOFTPLUS------
set.seed(1234567890)
winit <- runif(31, -1, 1)
nn_soft <- neuralnet(Sin ~ Var, data = tr, hidden = 10, startweights = winit,
                     act.fct = softplus_act)
#plot softplus
plot(tr, col = "black", main = "Softplus Activation", ylim = c(-1.5, 1.5), pch = 19)
points(te, col = "blue", cex = 0.5)
points(te[,1], predict(nn_soft, te), col = "red", cex = 1)
#reset plot layout
par(mfrow = c(1, 1))

#===== 4.3 Extrapolation =====
set.seed(1234567890)
Var <- runif(500, 0, 50)
data_new <- data.frame(Var, Sin = sin(Var))
#predict using NN from 4.1
pred_new <- predict(nn, data_new)
#Plot the results
#plot the true sine wave
plot(data_new$Var, data_new$Sin, col = "blue", cex = 0.5,
     xlim = c(0, 50), ylim = c(-10, 2),
     main = "NN Extrapolation (train 0-10, test 0-50)",
     xlab = "Var", ylab = "Sin")
#plot predictions (red)
points(data_new$Var, pred_new, col = "red", cex = 1)
#add vertical line to show where training stopped
abline(v = 10, col = "black", lwd = 2, lty = 2)
text(12, 1.5, "training limit", pos = 4)
legend("bottomright", legend = c("True Sin(x)", "Predicted"),
       col = c("blue", "red"), pch = c(1, 3))

#TASK 4.5 ===============
set.seed(1234567890)
#generate data
```

```
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin = sin(Var))
#train nn ( var ~ sin)
nn_inverse <- neuralnet(Var ~ Sin, data = mydata, hidden = 10, threshold = 0.1)
#Plot results
plot(mydata$Sin, mydata$Var, col = "black", cex = 0.5, pch = 19,
     main = "Predict x from Sin(x)",
     xlab = "Input: Sin(x)", ylab = "Target: x (Var)")
#Plot predictions (red)
preds <- predict(nn_inverse, mydata)
points(mydata$Sin, preds, col = "red", cex = 1, pch = 3)
legend("topleft", legend = c("True Data", "NN Prediction"),
       col = c("black", "red"), pch = c(19, 3))
```

Interpretation & Theory:

- **Activation Functions (4.2):**
  - **Linear:** Reduces the NN to simple Linear Regression. Cannot model the curve of Sine.
  - **ReLU:** Models Sine as piecewise linear segments. It fits well but has sharp ”corners”.
  - **Softplus:** A smooth approximation of ReLU. It fits the smooth Sine curve best.
- **Extrapolation (4.3):** Neural Networks are generally **poor extrapolators**. Outside the training range ($x > 10$), the activation functions (often Sigmoidal/Tanh in default settings) saturate, causing the prediction to flatten out into a constant value rather than continuing the periodic wave.
- **Inverse Task (4.5):** The function $x = \sin^{-1}(y)$ is **not a function** in this domain because it is one-to-many (e.g., $\sin(0) = 0, \sin(\pi) = 0, \sin(2\pi) = 0$). The Neural Network attempts to minimize MSE, so it learns the **conditional mean** of all possible x values for a given input, resulting in a flat line through the center of the data.

Exam 2024-08-28

Assignment 1: Trees, Logistic Regression & MLE

Problem Statement:
Data: *Australian-crabs.csv. Target: Sex, Features: measurements.*

1. ***Decision Trees (Model Selection):***
   - *Split 60% train / 40% test. Plot Train/Test Cross-Entropy vs Leaves.*
   - ***Report:*** *Optimal leaves, feature count, and why Cross-Entropy is a valid metric.*
   - ***Analysis:*** *Explain why branches in a 7-leaf tree lead to leaves with identical labels.*
2. ***Logistic Regression:***
   - *Train on **entire dataset**.*
   - *Predict prob for 1st obs. Calc change in prob if $w_{CW} = 0$ and $w_{BD} = 0$ (show math).*
   - *Compute F1 (Positive class=Male) using Loss Matrix:*

$$\begin{array}{rcc} & \text{Pred “Male”} & \text{Pred “Female”} \\ \text{True “Male”} & 0 & 1 \\ \text{True “Female”} & 10 & 0 \end{array}$$

   - ***Q:*** *Is F1 or Accuracy more relevant?*
3. ***MLE (Normal Regression):***
   - ***Model:*** $RW \sim \mathcal{N}(\mu = w_0 + w_1 FL, \sigma^2 = 0.1FL - 0.5)$.
   - *Implement minus log-likelihood. Optimize (BFGS, start=(0,0)) on entire data.*
   - *Compute prediction interval for 1st observation.*

```
# ASSIGNMENT 1 (2024-08-28)
# Load necessary library for Decision Trees
library(tree)

# 0. DATA PREPARATION
```

```r
# 1. Load Data
data <- read.csv("Australian-crabs.csv")
# 2. Fix Column Names (Ensure target is named 'sex')
# This line renames 'Sex', 'SEX', etc. to 'sex' to match your file exactly.
colnames(data)[tolower(colnames(data)) == "sex"] <- "sex"
# 3. Convert all character columns to factors
# The 'tree' package requires categorical variables to be factors, not strings.
# This prevents the "NAs introduced by coercion" error.
data[sapply(data, is.character)] <- lapply(data[sapply(data, is.character)], as.factor)
# Verify structure
str(data)

# TASK 1: DECISION TREES (Model Selection)
# 1. Split Data (60% Train / 40% Test)
set.seed(12345)
n <- nrow(data)
id <- sample(1:n, floor(0.6 * n))
train_data <- data[id, ]
test_data <- data[-id, ]
# 2. Fit the initial large tree
# Target: 'sex', Features: all others (.)
tree_model <- tree(sex ~ ., data = train_data)
# 3. Perform Pruning Sequence using Deviance (Cross-Entropy)
cv_tree <- prune.tree(tree_model, method = "deviance")
# 4. Calculate Test Cross-Entropy manually for each tree size
test_deviance <- numeric(length(cv_tree$size))
for(i in 1:length(cv_tree$size)) {
  k <- cv_tree$size[i]
  # Prune the tree to 'k' leaves
  pruned_tree <- prune.tree(tree_model, best = k, method = "deviance")
  # Predict on Test Data
  # HANDLE POTENTIAL ERROR: If tree is pruned to root only ("singlenode")
  if (inherits(pruned_tree, "singlenode")) {
    # If single node, predict the overall probability from training data for all rows
    train_probs <- prop.table(table(train_data$sex))
    preds <- matrix(rep(train_probs, each = nrow(test_data)),
                    nrow = nrow(test_data), byrow = FALSE)
    colnames(preds) <- names(train_probs)
  } else {
    # Normal prediction
    preds <- predict(pruned_tree, newdata = test_data, type = "vector")
  }
  # Calculate Deviance: -2 * sum( log( probability of true class ) )
  log_likelihood <- 0
  for (row in 1:nrow(test_data)) {
    true_cls <- as.character(test_data$sex[row]) # e.g., "Male"
    prob <- preds[row, true_cls]                 # Prob assigned to "Male"
    # Add epsilon to avoid log(0)
    log_likelihood <- log_likelihood + log(prob + 1e-15)
  }
  test_deviance[i] <- -2 * log_likelihood
}
# 5. Plot Training vs Test Cross-Entropy
y_lims <- range(c(cv_tree$dev, test_deviance))
plot(cv_tree$size, cv_tree$dev, type = "b", col = "blue", pch = 19, ylim = y_lims,
     xlab = "Number of Leaves", ylab = "Cross-Entropy (Deviance)",
     main = "Tree Complexity: Train vs Test Deviance")
lines(cv_tree$size, test_deviance, type = "b", col = "red", pch = 19)
legend("topright", legend = c("Train", "Test"), col = c("blue", "red"), lty = 1, pch = 19)
# 6. Report Optimal Number of Leaves
opt_leaves <- cv_tree$size[which.min(test_deviance)]
cat("Optimal number of leaves:", opt_leaves, "\n")
```

```r
# Inspect the optimal tree
final_tree <- prune.tree(tree_model, best = opt_leaves, method = "deviance")
summary(final_tree)

# TASK 2: LOGISTIC REGRESSION & F1-SCORE -------------------
# 1. Fit Logistic Regression on Entire Dataset
model_logit <- glm(sex ~ ., data = data, family = "binomial")
# 2. Predict Probability for First Observation
obs1 <- data[1, ]
prob_obs1 <- predict(model_logit, newdata = obs1, type = "response")
cat("Original predicted probability for Obs 1:", prob_obs1, "\n")
# 3. Manual Calculation (Setting CW and BD parameters to zero)
# Get coefficients
beta <- coef(model_logit)
# Zero out CW and BD
# Note: Check exact variable names in 'beta' if this fails.
# Usually they are "CW" and "BD" if numeric.
beta_mod <- beta
beta_mod["CW"] <- 0
beta_mod["BD"] <- 0
# Create feature vector for obs 1 (including Intercept)
x_vec <- model.matrix(sex ~ ., data = obs1)[1, ]
# Calculate linear predictor (z) and sigmoid probability
z_new <- sum(x_vec * beta_mod)
prob_new <- 1 / (1 + exp(-z_new))
cat("Updated probability (CW=0, BD=0):", prob_new, "\n")
cat("Difference:", prob_new - prob_obs1, "\n")
# 4. F1-Score with Asymmetric Loss Matrix
# Cost Ratio implies Threshold p > 10/11
threshold <- 10/11
cat("Optimal Threshold:", threshold, "\n")
# Predict on all data
probs_all <- predict(model_logit, type = "response")
# Identify which class corresponds to probability 1
# usually levels(data$sex)[2]
pos_class <- levels(data$sex)[2]
neg_class <- levels(data$sex)[1]
cat("Positive class (Prob > threshold):", pos_class, "\n")
# Classify
preds_class <- ifelse(probs_all > threshold, pos_class, neg_class)
preds_class <- factor(preds_class, levels = levels(data$sex))
# Confusion Matrix
cm <- table(Predicted = preds_class, Actual = data$sex)
print(cm)
# Calculate F1 for the "Positive" class (e.g., Male)
# Adjust indices [pos_class, pos_class] to match your data levels!
TP <- cm[pos_class, pos_class]
FP <- cm[pos_class, neg_class]
FN <- cm[neg_class, pos_class]
precision <- TP / (TP + FP)
recall <- TP / (TP + FN)
f1 <- 2 * (precision * recall) / (precision + recall)
cat("F1 Score:", f1, "\n")

# TASK 3: OPTIMIZATION (MLE) ------------------------
# 1. Define Minus Log-Likelihood Function
# Model: RW ~ Normal(mu = w0 + w1*FL, sigma^2 = 0.1*FL - 0.5)
nll_func <- function(theta, data) {
  w0 <- theta[1]
  w1 <- theta[2]
  y_val <- data$RW
  x_val <- data$FL
  # Define moments
  mu <- w0 + w1 * x_val
  sigma2 <- 0.1 * x_val - 0.5
  # Variance constraint check
  if (any(sigma2 <= 0)) return(Inf)
  # Log-likelihood
  ll <- -0.5 * log(2 * pi) - 0.5 * log(sigma2) - (y_val - mu)^2 / (2 * sigma2)
  # Return negative sum
  return(-sum(ll))
}
# 2. Optimization
start_params <- c(0, 0)
opt_res <- optim(par = start_params, fn = nll_func, data = data, method = "BFGS")
cat("Optimal Parameters (w0, w1):", opt_res$par, "\n")
# 3. Prediction Interval for Obs 1
w0_opt <- opt_res$par[1]
w1_opt <- opt_res$par[2]
fl_obs1 <- data$FL[1]
mu_pred <- w0_opt + w1_opt * fl_obs1
sigma2_pred <- 0.1 * fl_obs1 - 0.5
sigma_pred <- sqrt(sigma2_pred)
lower <- mu_pred - 1.96 * sigma_pred
upper <- mu_pred + 1.96 * sigma_pred
cat("95% Prediction Interval for Obs 1: [", lower, ", ", upper, "]\n")
```

Assignment 1 Analysis:

- **Why Cross-Entropy?** It is differentiable (unlike error rate) and heavily penalizes confident but wrong predictions ($p \approx 0$ or 1).
- **Duplicate Labels in Leaves:** Occurs when the class region is non-contiguous. The tree must split the feature space into separate boxes to capture disjoint regions of the same class.
- **LogReg Sensitivity:** Zeroing weights removes their contribution to the log-odds z. $P_{new} = \sigma(z_{old} - w_{CW}x_{CW} - w_{BD}x_{BD})$.
- **F1 vs Accuracy:** **F1** is relevant. The loss matrix is asymmetric (Cost 10 vs 0), making False Negatives much worse. Accuracy treats all errors equally.

Assignment 2: Neural Networks (Backprop & Dropout)

Problem Statement:
Data: *Synthetic. $x \sim U[-4, 4]$, $y = \sin(x)$ (500 pts).*

1. ***Implementation (Regression):***
   - ***Architecture:*** *1 Hidden Layer (2 units, Sigmoid $h(z)$), Output (Linear), Loss $J = (y - z^{(2)})^2$.*
   - ***Settings:*** *SGD (Batch=1), 100,000 iterations, $\gamma = 0.01$.*
   - ***Formulas:***
     - *Fwd: $z^{(1)} = W^{(1)}x + b^{(1)}$, $q^{(1)} = h(z^{(1)})$, $z^{(2)} = W^{(2)}q^{(1)} + b^{(2)}$.*
     - *Bwd: $dz^{(2)} = -2(y - z^{(2)})$, $dq^{(1)} = W^{(2)T}dz^{(2)}$, $dz^{(1)} = dq^{(1)} \odot h'(z^{(1)})$.*
     - *Grads: $dW^{(l)} = dz^{(l)}q^{(l-1)T}$, $db^{(l)} = dz^{(l)}$.*
   - *Plot MSE evolution and final prediction curve vs data.*
2. ***Dropout Regularization:***
   - *Incorporate dropout into backpropagation.*
   - *Run with dropout rates $1 - r \in \{0, 0.01, 0.05\}$ (keep prob $r \in \{1, 0.99, 0.95\}$).*
   - *Comment on results.*

```
set.seed(12345)
#generte training data
x <- runif(500, -4, 4)
x <- sin(x)
dat <- cbind(x, y)
plot(dat, main = "Training Data: y = sin(x)")
```

```
#activation functions!
h <- function(z) {
  return(1 / (1+exp(-z)))
}
hprime <- function(z) {
  #derivative of sigmoid
  return(h(z) * (1-h(z)))
}
#define prediction function, uses the trained weights (w1,b1,w2,b2)
yhat <- function(x_val){
  q0 <- x_val
  #layer 1
  z1 <- w1 %*% q0 + b1
  q1 <- h(z1)
  #layer 2
  z2 <- w2 %*% q1 + b2
  return(as.numeric(z2))
}
#define MSE func
MSE <- function(){
  preds <- sapply(dat[,1], yhat)
  return(mean((dat[,2] - preds)^2))
}
#main training function with dropout
#wrap training loop in a function to test droupout rates
train_network <- function(dropout_rate){
  cat("Training with Dropout Rate (1-r:", dropout_rate, "\n")
  #init parameters randomly
  #network structure: 1 input -> 2 hidden units -> 1 output
  #w1: 2x1 matrix, b1: 2x1 matrix
  #w2: 1x2 matrix, b2: 1x1 matrix
  #we use <<- to assign to gloal environment so yhat() can access
  w1 <<- matrix(runif(2, -0.1, 0.1), nrow = 2, ncol = 1)
  b1 <<- matrix(runif(2, -0.1, 0.1), nrow = 2, ncol = 1)
  w2 <<- matrix(runif(2, -0.1, 0.1), nrow = 1, ncol = 2)
  b2 <<- matrix(runif(1, -0.1, 0.1), nrow = 1, ncol = 1)
  gamma <- 0.01 #learning rate
  res <- numeric(0) #store mse history
  #calculate retention prob r
  r <- 1 - dropout_rate
  #training loop (100,000 iters)
  for(i in 1:100000){
    #store mse every 1000 iters
    if(i %% 1000 == 0){
      res <- c(res, MSE())
    }
    #select random training point
    j <- sample(1:nrow(dat), 1)
    q0 <- dat[j, 1]
    target <- dat[j, 2]
    #forward propagation
    z1 <- w1 %*% q0 + b1
    q1 <- as.matrix(h(z1)) #output hidden layer
    #apply dropout
    #inverted dropout: mask the units and scale by 1/r
    if(dropout_rate > 0){
      mask <- rbinom(n=2, size = 1, prob = r)
      q1 <- (q1 * mask) / r
    } else{
      mask <- c(1, 1) #n odropout, keep al
    }
    z2 <- w2 %*% q1 + b2 #outbut final layer
    #backward propagation
    #dz2 = -2 * (y - z2)
    dz2 <- -2 * (target - z2)
    #dq1 = W2_transpose * dz2
    dq1 <- t(w2) %*% dz2
    #dropout backprop
    #apply thr mask to gradient aswell
    if(dropout_rate > 0){
      dq1 <- (dq1 * mask) / r
    }
    #dz1 = dq1 * h'(Z1)
    dz1 <- dq1 * hprime(z1)
    # --- Compute Gradients ---
    dW2 <- dz2 %*% t(q1)
    db2 <- dz2
    dW1 <- dz1 %*% t(q0)
    db1 <- dz1
    # --- Parameter Updating  ---
    w2 <<- w2 - gamma * dW2
    b2 <<- b2 - gamma * db2
    w1 <<- w1 - gamma * dW1
    b1 <<- b1 - gamma * db1
  }
  # --- Results & Plotting ---
  # Plot MSE evolution
  plot(res, type = "l", main = paste("MSE History (Dropout =", dropout_rate, ")"),
       xlab = "Time (x1000 iter)", ylab = "MSE")
  # Plot fit against data [cite: 168-169]
  plot(dat, main = paste("Final Fit (Dropout =", dropout_rate, ")"))
  # Calculate predictions for sorting (to draw a line)
  x_seq <- seq(-4, 4, length.out = 200)
  preds_seq <- sapply(x_seq, yhat)
  lines(x_seq, preds_seq, col = "red", lwd = 2)
  cat("Final MSE:", tail(res, 1), "\n")
}
# 6. Run for the requested dropout rates
# Dropout 0 (r=1)
train_network(0)
# Dropout 0.01 (r=0.99)
train_network(0.01)
# Dropout 0.05 (r=0.95)
train_network(0.05)
```

Assignment 2 Analysis (Dropout):

- **Dropout Mechanism:** Randomly zeroing neurons prevents co-adaptation (neurons relying on specific peers), acting as an ensemble of sub-networks to reduce overfitting.
- **Results ($r = 1, 0.99, 0.95$):** Lower r (more dropout) increases regularization. $r = 1$: Baseline/Overfit. $r = 0.95$: Smoother fit, but risk of underfitting if capacity is low (only 2 hidden units).

Exam 2024-01-12

Assignment 1: Lasso Regression & Custom Optimization

Problem Statement:
Data: *Bikes.csv. Target: Rented Bike Count, Features: All others.*

1. ***Lasso Regression:***
   - *Split 70% train / 30% test.* ***Scale*** *data.*
   - *Fit LASSO (CV). Plot CV Error vs Penalty. Find optimal penalty.*
   - ***Interpret:*** *Bias-Variance tradeoff.*
   - ***Report:*** *Equation for lambda.1se model (scaled vars).*
2. ***Prediction Interval:***
   - *Use lambda.1se model. Estimate noise variance σ^2 using Training MSE.*
   - *Compute 95% PI for the* ***1st test observation****.*
   - ***Explain:*** *Assumptions required for this computation.*
3. ***Custom Optimization (BFGS):***
   - ***New Model:*** $DewPoint \sim Humidity + Visibility$ *(No intercept).*
   - ***Loss:*** $L(y, \hat{y}) = |y - \hat{y}|$ *(Mean Absolute Error).*
   - *Optimize using* ***BFGS****. Plot Cost & Test Error vs Iterations.*
   - ***Analysis:*** *Is early stopping needed? Report optimal iteration.*
   - ***Plots:*** *3 Scatter plots of (Humidity, Visibility) colored by: (a) True Target, (b) Optimal Model Preds, (c) 5-iteration Model Preds. Compare complexity/quality.*

```r
#----------- 1.1 -------
library(glmnet)
#load dataset
data <- read.csv("Bikes.csv")
#define target variable (rentedbieks)
target_var <- "RentedBikes"
#split data into trainng 70% and test 30% sets
set.seed(12345)
#calculate total number of rows in dataset
n <- nrow(data)
#randomly sample indices for training set
#floor(0.7 * n) calcs 70% of total rows,
train_indices <- sample(1:n, floor(0.7 * n))
#create "train_data" dataframe using selected indices
train_data <- data[train_indices, ]
#create test_data dataframe using remaining indiced
test_data <- data[-train_indices, ]
#create formula object. "rentedbikes depends on (~.) all other variables.
#the -1 tells R to remove the standard intercept, glmnet adds it on its own later
formula_def <- as.formula(paste(target_var, "~ . -1"))
#model.matrix converts dataframe into numeric matrix required by glmnet
X_train <- model.matrix(formula_def, data = train_data)
#extract target variable column (y) from training data
y_train <- train_data[[target_var]]
#scale data
#assignment asks for "scaled appropriately" and the equation for scaled variables
#scale() subtracts the mean and dided by standard deviation
#scale the feature matrix (x)
X_train_scaled <- scale(X_train)
#scale target vector (y)
y_train_scaled <- scale(y_train)
#LASSO
lasso_cv <- cv.glmnet(X_train_scaled, y_train_scaled, alpha = 1)
#plot
plot(lasso_cv)
```

```
title("CV Error vs Log(Lambda)", line = 2.5)
#identify optimal lambda value using one standard error rule (1se)
best_lambda <- lasso_cv$lambda.1se
#print optimal lambda
cat("optimal lambda (1se):", best_lambda, "\n")
#extract beta values (coefficients) for them model using optimal lambda
#the s argument specifies which lambda to use
final_coefs <- coef(lasso_cv, s = "lambda.1se")
print("coef for the scaled Lasso model")
print(final_coefs)

#---------- 1.2: Prediction Interval ---------------
#Calc MSE on Training data, use model to pred values for training set first
#newx expects the scaled matrix we made in 1.1 (X_train_scaled)
train_predictions <- predict(lasso_cv, newx = X_train_scaled, s="lambda.1se")
#calc MSE, estimate for error variance (sigma^2)
mse_train <- mean((y_train_scaled - train_predictions)^2)
cat("Estimated Error Variance (MSE from training):", mse_train, "\n")
#prepare the First Test Observation
#create matrix for test data just like in training
#create raw test matrix
X_test_raw <- model.matrix(formula_def, data=test_data)
#retrieve scaling parameters from traiing step
train_center <- attr(X_train_scaled, "scaled:center")
train_scale <- attr(X_train_scaled, "scaled:scale")
#scale the test matrix using training statistics
X_test_scaled <- scale(X_test_raw, center = train_center, scale = train_scale)
#extract ONLY first row (first observation) for prediction
first_obs_scaled <- X_test_scaled[1, , drop = FALSE]
#predict the target value for the first observation
y_pred_scaled <- predict(lasso_cv, newx = first_obs_scaled, s = "lambda.1se")
cat("predicted scaled value:", y_pred_scaled, "\n")
#calculate 95% prediction interval, formula: Prediction +/- 1.96 * sqrt(MSE)
#use 1.96 bcs it corresponds to 97,5th percentile of normal distrib.
margin_of_error <- 1.96 * sqrt(mse_train)
lower_bound <- y_pred_scaled - margin_of_error
upper_bound <- y_pred_scaled + margin_of_error
cat("95% Predic interval (scaled): [", lower_bound, ", ", upper_bound, "]\n")

#--------------------- 1.3---------------------
library(ggplot2)
# --- 1. Data & Skalning ---
# Anvand 'train_data' och 'test_data' fran del 1.1
vars <- c("Humidity", "Visibility")
target_13 <- "Dew.point.temperature"
# Plocka ut specifika kolumner och skala dem
# Vi använder as.matrix() for att vara sakra pa att det fungerar med optimeringen senare
X_train_13 <- scale(as.matrix(train_data[, vars]))
y_train_13 <- scale(train_data[[target_13]])
# Spara skalningsparametrar från traning
center_vals <- attr(X_train_13, "scaled:center")
scale_vals <- attr(X_train_13, "scaled:scale")
y_center <- attr(y_train_13, "scaled:center")
y_scale <- attr(y_train_13, "scaled:scale")
# Skala testdata med traningsdatans parametrar
X_test_13 <- scale(as.matrix(test_data[, vars]), center = center_vals, scale = scale_vals)
y_test_13 <- scale(test_data[[target_13]], center = y_center, scale = y_scale)
# --- 2. Kostnadsfunktion (MAE) & Optimering ---
# MAE Cost Function
mae_loss <- function(par, X, y) {
  mean(abs(y - X %*% par))
}
```

```r
results <- data.frame(Iter = 1:100, Train = NA, Test = NA)
for(i in 1:100) {
  # Optimera med max 'i' iterationer
  opt <- optim(par = c(0,0), fn = mae_loss, X = X_train_13, y = y_train_13,
               method = "BFGS", control = list(maxit = i))
  results$Iter[i] <- i
  results$Train[i] <- opt$value
  results$Test[i] <- mae_loss(opt$par, X_test_13, y_test_13)
}
# Hitta optimal iteration
opt_iter <- which.min(results$Test)
cat("Optimal Iteration:", opt_iter, "\n")
# --- 3. Plottar ---
# Plot 1: Iterationshistorik
plot(results$Iter, results$Train, type="l", col="blue",
     ylim=range(c(results$Train, results$Test)),
     xlab="Iteration", ylab="MAE", main="Optimization History")
lines(results$Iter, results$Test, col="red")
legend("topright", legend=c("Train", "Test"), col=c("blue", "red"), lty=1)
# Forbered data for scatter-plots
# Kor optimering igen for att fa fram parametrar
par_opt <- optim(c(0,0), mae_loss, X=X_train_13, y=y_train_13, method="BFGS", control=list(maxit=
    ↪ opt_iter))$par
par_5   <- optim(c(0,0), mae_loss, X=X_train_13, y=y_train_13, method="BFGS", control=list(maxit
    ↪ =5))$par
plot_data <- data.frame(
  Hum = X_train_13[,"Humidity"],
  Vis = X_train_13[,"Visibility"],
  Original = as.numeric(y_train_13),
  Pred_Opt = (X_train_13 %*% par_opt)[,1],
  Pred_5   = (X_train_13 %*% par_5)[,1]
)
# Funktion for att rita plottar
my_scatter <- function(data, color_col, title) {
  ggplot(data, aes(x=Hum, y=Vis, color=color_col)) +
    geom_point(alpha=0.6) + scale_color_viridis_c() + ggtitle(title) + theme_minimal()
}
# Skapa de tre plottarna
p1 <- my_scatter(plot_data, plot_data$Original, "a) Original Target")
p2 <- my_scatter(plot_data, plot_data$Pred_Opt, paste("b) Predicted (Iter", opt_iter, ")"))
p3 <- my_scatter(plot_data, plot_data$Pred_5, "c) Predicted (Iter 5)")
# Visa dem
print(p1)
print(p2)
print(p3)
```

Assignment 1 Analysis:

- **Lasso & Bias-Variance:** λ controls complexity. High λ = High Bias / Low Variance (Underfitting). Low λ = Low Bias / High Variance (Overfitting). We choose 'lambda.1se' (within 1 SE of min error) to prefer a sparser (simpler) model that generalizes better[cite: 141].
- **PI Assumptions:** To use MSE_{train} for prediction intervals ($\hat{y} \pm 1.96\sqrt{MSE}$), we assume errors are independent, homoscedastic (constant variance), and normally distributed: $\epsilon \sim \mathcal{N}(0, \sigma^2)$.
- **Custom Loss (MAE):** $L = |y - \hat{y}|$ (L1 norm) is more robust to outliers than MSE.
- **Early Stopping:** Needed if Test Error begins to increase while Training Error decreases (overfitting). If Test Error plateaus/decreases alongside Train Error, it is not needed.

Assignment 2: Neural Networks (Dropout) & Perceptron

Problem Statement:

1. ***Neural Networks (Dropout):***
   - *Base Code: TDDE01 January2023.R (Backprop for Regression).*
   - ***Task:*** *Incorporate* ***Dropout****.*
   - *Run with keep probabilities $r \in \{1, 0.99, 0.95\}$ (Dropout rates $0, 0.01, 0.05$).*
   - *Comment on results.*
2. ***Perceptron Implementation:***
   - ***Task:*** *Binary classification ($t \in \{-1, 1\}$). 2D Data (code provided).*
   - ***Update Rule:*** $w^{(new)} = w^{(old)} + \alpha \sum (t_n - y(w, x_n)) x_n$.
   - ***Prediction:*** $y(w, x) = 1$ *if* $w^T x \geq 0$*, else* -1.
   - *Run 100 iterations,* $\alpha = 0.0001$.
   - ***Plot:*** *Misclassification rate vs Iterations.*
   - ***Explain:*** *When does Perceptron work best and why?*

```r
#EXERCISE 1, BASE CODE TDDE01 JANUARY2023.R (BACKDROP FOR REGRESSON
    set.seed(1234)
# 1. Setup Data
# produce the training data in dat
x <- runif(500, -4, 4)
y <- sin(x)
dat <- cbind(x, y)
# Plot original data once
plot(dat, main = "Original Data")
# 2. Define Helper Functions
h <- function(z) {
  # activation function (sigmoid)
  return(1 / (1 + exp(-z)))
}
hprime <- function(z) {
  # derivative of the activation function (sigmoid)
  return(h(z) * (1 - h(z)))
}
# Note: yhat uses the global variables w1, b1, w2, b2.
# Because we use Inverted Dropout, we do not need to modify yhat.
yhat <- function(x) {
  # prediction for point x
  q0 <- x
  z1 <- w1 %*% q0 + b1
  q1 <- as.matrix(apply(z1, 1, h), nrow = 2, ncol = 1)
  z2 <- w2 %*% q1 + b2
  return(z2)
}
MSE <- function() {
  # mean squared error
  res <- NULL
  for (i in 1:nrow(dat)) {
    res <- c(res, (dat[i, 2] - yhat(dat[i, 1]))^2)
  }
  return(mean(res))
}
# 3. Define Training Function with Dropout
train_network <- function(dropout_rate) {
  # Initialize parameters freshly for each run
  # We use <<- to update the global variables so yhat() can see them
  w1 <<- matrix(runif(2, -.1, .1), nrow = 2, ncol = 1)
  b1 <<- matrix(runif(2, -.1, .1), nrow = 2, ncol = 1)
  w2 <<- matrix(runif(2, -.1, .1), nrow = 1, ncol = 2)
  b2 <<- matrix(runif(1, -.1, .1), nrow = 1, ncol = 1)
  gamma <- 0.01
  res <- NULL
  # Calculate retention probability (r)
  keep_prob <- 1 - dropout_rate

  print(paste("Training with Dropout Rate:", dropout_rate))
  for (i in 1:100000) {
    if (i %% 1000 == 0) {
      res <- c(res, MSE())
    }
    # --- Forward propagation ---
    j <- sample(1:nrow(dat), 1)
    q0 <- dat[j, 1]
    z1 <- w1 %*% q0 + b1
    q1 <- as.matrix(apply(z1, 1, h), nrow = 2, ncol = 1)
    # --- APPLY DROPOUT ---
    # Create mask: 1 with probability (1-rate), 0 with probability (rate)
    if (dropout_rate > 0) {
      mask <- rbinom(n = 2, size = 1, prob = keep_prob)
      # Inverted Dropout: Zero out units AND scale up by 1/keep_prob
      # This ensures the expected value of q1 remains the same for the next layer
      q1 <- (q1 * mask) / keep_prob
    } else {
      # No dropout needs a mask of 1s for the backward pass math to hold conceptually
      mask <- c(1, 1)
    }
    z2 <- w2 %*% q1 + b2

    # --- Backward propagation ---
    dz2 <- -2 * (dat[j, 2] - z2)
    # Backprop through weights
    dq1 <- t(w2) %*% dz2
    # Backprop through Dropout (Gradient only flows where mask was 1)
    if (dropout_rate > 0) {
      dq1 <- (dq1 * mask) / keep_prob
    }
    dz1 <- dq1 * hprime(z1)
    dw2 <- dz2 %*% t(q1)
    db2 <- dz2
    dw1 <- dz1 %*% t(q0)
    db1 <- dz1
    # --- Parameter updating ---
    w2 <<- w2 - gamma * dw2
    b2 <<- b2 - gamma * db2
    w1 <<- w1 - gamma * dw1
    b1 <<- b1 - gamma * db1
  }
  # Plotting the loss curve
  plot(res, type = "l", main = paste("Loss Curve (Dropout =", dropout_rate, ")"))
  # Plotting the fit
  plot(dat, main = paste("Fit with Dropout =", dropout_rate))
  points(dat[, 1], lapply(dat[, 1], yhat), col = "red")
  print(paste("Final MSE:", tail(res, 1)))
}
# 4. Run the requested exercises
# Dropout rate 0
train_network(0)
# Dropout rate 0.01
train_network(0.01)
```

```
# Dropout rate 0.05
train_network(0.05)
```

```
    # ---------------------------------------------------------
# Assignment 2, Exercise 2: Perceptron Algorithm
# 1. Data Generation (From exam instructions)
set.seed(1234)
x <- array(NA, dim=c(100,2))
t <- array(NA, dim=c(100))
x[,1] <- runif(100, 0, 3)
x[,2] <- runif(100, 0, 9)
# Target t is -1 if under the parabola x^2, else +1
t <- ifelse(x[,2] < (x[,1])^2, -1, 1)
# Visualization (Optional)
plot(x[,1], x[,2], col=ifelse(t==1, "blue", "red"), pch=19,
     main="Data Distribution", xlab="x1", ylab="x2")
# 2. Perceptron Algorithm Implementation
# Initialize weights randomly
w <- runif(2, -0.1, 0.1)
# Hyperparameters
alpha <- 0.0001
max_iter <- 100
misclass_rates <- numeric(max_iter)
# Training Loop
for(k in 1:max_iter){
  # A. Compute predictions
  # y(w,x) = 1 if w^T * x >= 0, else -1
  linear_output <- x %*% w
  y_pred <- ifelse(linear_output >= 0, 1, -1)
  # B. Calculate Misclassification Rate
  error_count <- sum(y_pred != t)
  misclass_rates[k] <- error_count / length(t)
  # C. Update Weights
  # Formula: w_new = w_old + alpha * sum((t - y) * x)
  diff <- t - y_pred
  # Vi använder as.vector() för att konvertera 'diff' från en array till en
  # vanlig vektor. Detta tillåter R att multiplicera den med matrisen 'x' korrekt.
  gradient_component <- colSums(as.vector(diff) * x)
  # Update weights
  w <- w + alpha * gradient_component
}
# 3. Plot Misclassification Rate
plot(1:max_iter, misclass_rates, type="l", col="blue", lwd=2,
     main="Perceptron Misclassification Rate",
     xlab="Iteration", ylab="Error Rate")
# Print final results
cat("Final Error Rate:", tail(misclass_rates, 1), "\n")
cat("Final Weights:", w, "\n")
```

Assignment 2 Analysis:

- **Dropout Results:** $r = 1$ (No dropout) risks overfitting. $r = 0.99/0.95$ introduces noise, forcing the network to learn robust features (regularization), typically smoothing the prediction curve.
- **Perceptron Performance:** Works best when data is **linearly separable**.
- **Why?** The Perceptron Convergence Theorem guarantees it will find a solution in finite steps *only* if a linear hyperplane exists that perfectly separates the classes. If data is non-linear (like the parabola in the code $x_2 < x_1^2$), it will never converge and weights will oscillate.

Exam 2023-03-17

Assignment 1: KNN, PCA Optimization

Problem Statement:
Data: *tecator.csv. Target: Fat/Protein, Features: Channels.*

1. ***KNN (Bias-Variance):***
   - *Split 50% train / 50% test.*
   - *Train KNN ($k = 1, \ldots, 30$) with Target = Fat, Features = Channels.*
   - ***Plot:*** *Train/Test MSE vs k. Interpret Bias-Variance tradeoff.*
   - ***Report:*** *Optimal k and corresponding Train/Test MSE.*
   - *Note: Use correct kernel in kknn for standard KNN.*
2. ***PCA + KNN:***
   - *Perform PCA on Channels (unscaled). Get scores.*
   - *Split scores 50/50 (same indices).*
   - *Train KNN (Target = Fat, Features = **First 10 PCs**, k = optimal from Step 1).*
   - ***Compare:*** *MSEs vs Step 1. Why the difference?*
3. ***Optimization (Conjugate Gradient):***
   - ***Scale** data. Model: $Protein \sim w_0 + w_1 Fat + w_2 Fat^2 + w_3 Fat^3$.*
   - *Optimize using **CG** (start $\vec{0}$). Plot log(MSE) vs Iterations.*
   - ***Analysis:*** *Is early stopping needed?*
   - ***Plot:*** *Overlay model predictions on data for Iterations = 20 vs 200. Compare.*

```r
library(kknn)
# 1. Load Data
# Using read.csv2 is often better for files with comma decimals, but we can fix manually too.
data <- read.csv("tecator.csv")
# data has commas (",") instead of dots ("."), making them text.
# We must convert them to numbers.
# Helper function to fix comma decimals
fix_comma_decimal <- function(x) {
  if (is.character(x)) {
    x <- sub(",", ".", x) # Replace comma with dot
    return(as.numeric(x)) # Convert to number
  }
  return(x)
}
# Apply to all columns
data[] <- lapply(data, fix_comma_decimal)
# Remove any rows where 'Fat' is NA (judging by your output, row 1 might be NA)
data <- data[!is.na(data$Fat), ]
# Verify structure - you should see 'num' (numeric) now, not 'chr'
str(data)
# 2. Split Data (50/50)
set.seed(12345)
n <- nrow(data)
id <- sample(1:n, floor(0.5 * n))
train_data <- data[id, ]
test_data <- data[-id, ]
# 3. Prepare Subsets
channel_cols <- grep("Channel", colnames(data), value = TRUE)
train_sub <- train_data[, c("Fat", channel_cols)]
test_sub <- test_data[, c("Fat", channel_cols)]
# 4. KNN Loop (Safe version)
# Ensure k doesn't exceed training size
k_max <- min(30, nrow(train_sub) - 1)
k_values <- 1:k_max
train_mse <- numeric(length(k_values))
```

```
test_mse <- numeric(length(k_values))
for(i in seq_along(k_values)) {
  k <- k_values[i]
  # Train and Predict on Training Data
  # Fat ~ . uses all other columns (Channels) as features
  model_train <- kknn(Fat ~ ., train = train_sub, test = train_sub, k = k, kernel = "rectangular"
    ↪ )
  preds_train <- model_train$fitted.values
  train_mse[i] <- mean((train_sub$Fat - preds_train)^2)
  # Train on Train, Predict on Test
  model_test <- kknn(Fat ~ ., train = train_sub, test = test_sub, k = k, kernel = "rectangular")
  preds_test <- model_test$fitted.values
  test_mse[i] <- mean((test_sub$Fat - preds_test)^2)
}
# 5. Plot Bias-Variance Tradeoff
y_lims <- range(c(train_mse, test_mse))
plot(k_values, test_mse, type = "b", col = "red", pch = 19, ylim = y_lims,
     xlab = "k (Number of Neighbors)", ylab = "MSE", main = "KNN Bias-Variance Tradeoff")
lines(k_values, train_mse, type = "b", col = "blue", pch = 19)
legend("topright", legend = c("Test MSE", "Train MSE"), col = c("red", "blue"), lty = 1, pch =
    ↪ 19)
# 6. Report Optimal Model
opt_idx <- which.min(test_mse)
cat("Optimal k:", k_values[opt_idx], "\n")
cat("Test MSE at optimal k:", test_mse[opt_idx], "\n")

# ----------------------------------------------------------
# TASK 2: PCA & KNN
# 1. Perform PCA on Original Channel Data
# Instruction: "Perform PCA on all Channel columns... without scaling"
# We use the cleaned 'data' object from Task 1 (where commas are fixed)
channel_cols <- grep("Channel", colnames(data), value = TRUE)
pca_data <- data[, channel_cols]
# Run PCA
pca_res <- prcomp(pca_data, scale. = FALSE)
# 2. Extract Scores (Coordinates)
# Instruction: Use "first 10 PC columns"
pca_scores <- as.data.frame(pca_res$x[, 1:10])
# Add the Target variable 'Fat' back to this PCA dataset
pca_scores$Fat <- data$Fat
# 3. Split Data (Using SAME indices as Task 1)
# We must use the 'id' variable created in Task 1
train_pca <- pca_scores[id, ]
test_pca <- pca_scores[-id, ]
# 4. Train KNN with Optimal k
# Use the 'opt_k' found in Task 1 (e.g., if opt_k was 4)
# If you don't have 'opt_k' in memory, set it manually (e.g., k = 4)
if(!exists("opt_k")) opt_k <- 4 # Fallback if Task 1 wasn't run in this session
cat("Training PCA-KNN with k =", opt_k, "\n")
# Train on Train, Predict on Train (to get Training MSE)
model_pca_train <- kknn(Fat ~ ., train = train_pca, test = train_pca, k = opt_k, kernel = "
    ↪ rectangular")
preds_pca_train <- model_pca_train$fitted.values
mse_pca_train <- mean((train_pca$Fat - preds_pca_train)^2)
# Train on Train, Predict on Test (to get Test MSE)
model_pca_test <- kknn(Fat ~ ., train = train_pca, test = test_pca, k = opt_k, kernel = "
    ↪ rectangular")
preds_pca_test <- model_pca_test$fitted.values
mse_pca_test <- mean((test_pca$Fat - preds_pca_test)^2)
# 5. Report Results
cat("--------------------------------------------------\n")
cat("Results for PCA-based KNN (10 PCs):\n")
cat("Training MSE:", mse_pca_train, "\n")
cat("Test MSE:", mse_pca_test, "\n")
cat("------------------------------------------------\n")
cat("Comparison with Original Model (Task 1):\n")
# Assuming 'test_mse' and 'opt_idx' exist from Task 1
if(exists("test_mse") & exists("opt_idx")) {
  cat("Original Test MSE:", test_mse[opt_idx], "\n")
  cat("Difference (PCA - Original):", mse_pca_test - test_mse[opt_idx], "\n")
}

# ---------------------------------------------------------
# TASK 1.3: OPTIMIZATION (Polynomial Regression)
# 1. Scale Data
# We select only the relevant variables: Fat (feature) and Protein (target)
# Note: Ensure 'Fat' is numeric (fixed in Task 1)
train_raw <- train_data[, c("Fat", "Protein")]
test_raw <- test_data[, c("Fat", "Protein")]
# Scale Training Data
train_scaled <- scale(train_raw)
# Scale Test Data using Training parameters (Standard practice)
# We extract the center (mean) and scale (sd) from the training set
train_center <- attr(train_scaled, "scaled:center")
train_scale <- attr(train_scaled, "scaled:scale")
test_scaled <- scale(test_raw, center = train_center, scale = train_scale)
# Convert to simple vectors/matrices for the cost function
X_train <- train_scaled[, "Fat"]
y_train <- train_scaled[, "Protein"]
X_test <- test_scaled[, "Fat"]
y_test <- test_scaled[, "Protein"]
# 2. Define Cost Function (MSE)
# Model: Protein = w0 + w1*Fat + w2*Fat^2 + w3*Fat^3
mse_cost <- function(theta, X, y) {
  w0 <- theta[1]
  w1 <- theta[2]
  w2 <- theta[3]
  w3 <- theta[4]
  # Calculate predictions
  preds <- w0 + w1*X + w2*(X^2) + w3*(X^3)
  # Return Mean Squared Error
  return(mean((y - preds)^2))
}
# 3. Optimization Loop (To track progress)
# We need to plot log(MSE) vs iteration number.
# optim() with 'CG' doesn't return per-iteration history by default.
# The standard way for this assignment is to run optim repeatedly with increasing 'maxit'.
max_iter <- 200
results <- data.frame(Iter = 1:max_iter, Train_logMSE = NA, Test_logMSE = NA)
# Initial parameters (Zero vector)
init_params <- c(0, 0, 0, 0)
for(i in 1:max_iter) {
  # Run optimization with a limit of 'i' iterations
  opt_res <- optim(par = init_params, fn = mse_cost, X = X_train, y = y_train,
                   method = "CG", control = list(maxit = i))
  # Calculate MSE for the parameters found at step 'i'
  # Training MSE
  mse_tr <- mse_cost(opt_res$par, X_train, y_train)
  results$Train_logMSE[i] <- log(mse_tr)
  # Test MSE
  mse_te <- mse_cost(opt_res$par, X_test, y_test)
  results$Test_logMSE[i] <- log(mse_te)
}
# 4. Plot log(MSE) vs Iteration
```

```
plot(results$Iter, results$Train_logMSE, type = "l", col = "blue", lwd = 2,
     ylim = range(c(results$Train_logMSE, results$Test_logMSE)),
     xlab = "Iteration Number", ylab = "log(MSE)",
     main = "CG Optimization: Early Stopping Check")
lines(results$Iter, results$Test_logMSE, col = "red", lwd = 2)
legend("topright", legend = c("Train log(MSE)", "Test log(MSE)"),
       col = c("blue", "red"), lwd = 2)
# 5. Compare Iteration 20 vs Iteration 200
# Get parameters for iter 20
opt_20 <- optim(par = init_params, fn = mse_cost, X = X_train, y = y_train,
                method = "CG", control = list(maxit = 20))
params_20 <- opt_20$par
# Get parameters for iter 200
opt_200 <- optim(par = init_params, fn = mse_cost, X = X_train, y = y_train,
                 method = "CG", control = list(maxit = 200))
params_200 <- opt_200$par
# Create a sequence for smooth plotting of the model curve
x_grid <- seq(min(X_train), max(X_train), length.out = 100)
# Predict for grid using both models
pred_20 <- params_20[1] + params_20[2]*x_grid + params_20[3]*(x_grid^2) + params_20[4]*(x_grid^3)
pred_200 <- params_200[1] + params_200[2]*x_grid + params_200[3]*(x_grid^2) + params_200[4]*(x_
    ↪ grid^3)
# Plot Training Data + Model Curves
plot(X_train, y_train, pch = 19, col = "gray",
     xlab = "Fat (Scaled)", ylab = "Protein (Scaled)",
     main = "Model Fit: Iteration 20 vs 200")
lines(x_grid, pred_20, col = "red", lwd = 2, lty = 2)   # Iter 20 (Dashed Red)
lines(x_grid, pred_200, col = "green", lwd = 2)         # Iter 200 (Solid Green)
legend("topright", legend = c("Iter 20", "Iter 200", "Data"),
       col = c("red", "green", "gray"), lty = c(2, 1, NA), pch = c(NA, NA, 19))
```

Assignment 1 Analysis:

- **KNN Bias-Variance:** Small k (complex model) $\rightarrow$ Low Bias, High Variance (Overfitting). Large k (simple model) $\rightarrow$ High Bias, Low Variance (Underfitting).
- **PCA vs Raw Data:** PCA compresses variance into orthogonal components. Using only the top 10 PCs likely improves KNN performance by filtering out noise features and mitigating the *curse of dimensionality* present in high-dimensional spectral data.
- **CG Optimization Complexity:** The number of iterations acts as a regularization parameter.
  - **20 Iterations:** Underfitted. The optimizer hasn't converged; the curve will look too flat/simple.
  - **200 Iterations:** Better fit. Captures the polynomial curvature (3rd degree).
  - **Early Stopping:** Necessary if the Test Error begins to rise while Training Error continues to decrease (sign of overfitting).

Assignment 2: SVM Kernel Methods

Problem Statement:
Data: *spam (from kernlab).*

1. ***SVM (Model Selection):***
   - *Use ksvm. Kernel: RBF (width 0.05). $C \in \{0.5, 1, 5\}$.*
   - *Select best model (any method). Estimate generalization error.*
   - *Show final code. Explain purpose of parameter C.*
2. ***Kernel Methods (Regression for Classification):***
   - *Modify lab code (Temperature prediction) to classify spam.*
   - *Treat class label as continuous (Regression).*
   - *Split 2/3 train, 1/3 test. Use **first 48 attributes**.*
   - *Use Gaussian Kernel. Test different widths.*

```
library(kernlab)
```

```r
data(spam)
# 2. Split Data (Using random 70/30 split for model selection)
set.seed(12345)
n <- nrow(spam)
id <- sample(1:n, floor(0.7 * n))
train_data <- spam[id, ]
test_data <- spam[-id, ]
#train models with different c values
#Kernel: Radial Basis (gaussian) with sigma = 0.05
C_values <- c(0.5, 1, 5)
results <- data.frame(C= C_values, Error = NA)
for(i in 1:length(C_values)){
  c_val <- C_values[i]
  #Train SVM
 # type="C-svc" is standard calssification
  model<- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
               kpar = list(sigma = 0.05), C = c_val)
  #predict on test data (validation)
  preds <- predict(model, test_data)
  #calculate missclassification error
  cm <- table(preds, test_data$type)
  accuracy <- sum(diag(cm)) / sum(cm)
  error_rate <- 1 - accuracy
  results$Error[i] <- error_rate
  cat("C =", c_val, "-> Error Rate:", error_rate, "\n")
}
#select best model
best_idx <- which.min(results$Error)
best_C <- results$C[best_idx]
best_error <- results$Error[best_idx]
cat("\nBest Model has C =", best_C, "with Error:", best_error, "\n")
#generalizaton error estimate
cat("Estimated Generalization Error:", best_error, "\n")

# =============================================================================
# ASSIGNMENT 2: PART 2
# 1. Prepare Data
spam_subset <- spam[, c(1:48, 58)]
spam_subset$type <- ifelse(spam_subset$type == "spam", 1, 0)
# Set the REQUIRED exam seed
set.seed(12345)
# Split 2/3 Training, 1/3 Testing
n <- nrow(spam_subset)
id <- sample(1:n, floor(2/3 * n))
train_k <- spam_subset[id, ]
test_k <- spam_subset[-id, ]
X_train <- as.matrix(train_k[, 1:48])
y_train <- train_k$type
X_test <- as.matrix(test_k[, 1:48])
y_test <- test_k$type
# 2. Define Kernel Function (Vectorized for speed)
predict_kernel <- function(X_train, y_train, X_test, h) {
  preds <- numeric(nrow(X_test))
  for(i in 1:nrow(X_test)) {
    test_point <- X_test[i, ]
    # Calculate distance to ALL training points at once
    diffs <- t(t(X_train) - test_point)
    dists_sq <- rowSums(diffs^2)
    weights <- exp(-dists_sq / h)
    if(sum(weights) < 1e-10) {
      preds[i] <- mean(y_train)
    } else {
      preds[i] <- sum(weights * y_train) / sum(weights)
    }
  }
  return(preds)
}
# 3. Run for different widths
# We take a RANDOM subset of 200 points from the test set for speed
# (This maintains the seed 12345 randomness flow)
test_indices <- sample(1:nrow(X_test), 200)
X_test_small <- X_test[test_indices, ]
y_test_small <- y_test[test_indices]
h_values <- c(10, 20, 50, 100)
cat("\n--- Kernel Classifier Results ---\n")
for(h in h_values) {
  probs <- predict_kernel(X_train, y_train, X_test_small, h)
  pred_class <- ifelse(probs > 0.5, 1, 0)
  acc <- mean(pred_class == y_test_small)
  cat("Width h =", h, "-> Accuracy:", acc, "\n")
}
```

Assignment 2 Analysis:

- **SVM Parameter C:** Controls the trade-off between maximizing the geometric margin and minimizing misclassification errors.
  - **High C:** High penalty for errors $\rightarrow$ "Hard" margin $\rightarrow$ Low Bias, High Variance (Overfitting risk).
  - **Low C:** Low penalty $\rightarrow$ "Soft" margin $\rightarrow$ High Bias, Low Variance (Smoother boundary).

Exam 2022-03-18

Assignment 1: PCA, Logistic Regression & Custom Optimization

Problem Statement:
Data: *geneexp.csv. Target: CellType (B-cell/T-cell). Clean zeros (remove columns containing zeros only).*

1. ***PCA Analysis:***
   - *Perform PCA. Report variation (first 2 PCs) & scaling importance.*
   - *Plot scores (PC1-PC2) colored by CellType. Is classification easy?*
2. ***Logistic Regression:***
   - ***Model A (All genes):*** *Report Confusion Matrix.*
   - ***Model B (2 PCs):*** *Report Probabilistic model.*
   - ***Theory:*** *Why is High p (predictors/features/variables), Low n (samples/data points) problematic for ML models?*
3. ***Custom Logistic Loss (BFGS):***
   - *Implement $J(\theta)$ (Code + Formula).*
   - *Optimize (BFGS, start $\vec{0}$, 50/50 split).*
   - *Plot Train/Test Cost vs Iterations (maxit=20). Check early stopping.*

```r
# ==============================================================================
# ASSIGNMENT 1 (2022-03-18) - TASK 1
# 1. Load Data
data <- read.csv("geneexp.csv")
# 2. Data Cleaning (Safer Method)
# Create a temporary subset of only gene columns (excluding 'CellType')
gene_vars <- data[, colnames(data) != "CellType"]
# Identify columns that are NOT all zeros
# We calculate the sum of absolute values; if sum > 0, the column has data.
keep_indices <- which(colSums(abs(gene_vars)) != 0)
# Create the cleaned dataset
# Combine 'CellType' with only the valid gene columns
```

```r
data_clean <- data.frame(CellType = data$CellType, gene_vars[, keep_indices])
# Verify the result
cat("Original number of columns:", ncol(data), "\n")
cat("Cleaned number of columns:", ncol(data_clean), "\n")
# 3. Perform PCA
# Perform PCA on all columns except the first one ('CellType')
# scale. = TRUE is required to normalize the variance across genes [cite: 89-90]
pca_res <- prcomp(data_clean[, -1], scale. = TRUE)
# 4. Explained Variance
summ <- summary(pca_res)
var_explained <- summ$importance[2, 1:2] # Extract proportion of variance for PC1 and PC2
total_var_2 <- sum(var_explained)
cat("Variance explained by PC1:", var_explained[1], "\n")
cat("Variance explained by PC2:", var_explained[2], "\n")
cat("Total variance (PC1+PC2):", total_var_2, "\n")
# 5. Plot Scores
# Create a dataframe for plotting
scores <- data.frame(PC1 = pca_res$x[, 1],
                     PC2 = pca_res$x[, 2],
                     CellType = data_clean$CellType)
# Plot
plot(scores$PC1, scores$PC2, col = as.factor(scores$CellType), pch = 19,
     xlab = paste0("PC1 (", round(var_explained[1]*100, 1), "%)"),
     ylab = paste0("PC2 (", round(var_explained[2]*100, 1), "%)"),
     main = "PCA of Gene Expression Data")
legend("topright", legend = levels(as.factor(scores$CellType)),
       col = 1:2, pch = 19)

# ================================
# ASSIGNMENT 1.2
# 1. Setup: Load and Clean Data (Required for Task 2 to work)
data <- read.csv("geneexp.csv")
# Identify columns that are NOT 'CellType'
gene_vars <- data[, colnames(data) != "CellType"]
# Find columns that are not all zeros
keep_indices <- which(colSums(abs(gene_vars)) != 0)
# Create clean dataset and ensure CellType is a FACTOR (Fixes "0 <= y <= 1" error)
data_clean <- data.frame(CellType = as.factor(data$CellType), gene_vars[, keep_indices])
# Perform PCA (Required for Model B)
pca_res <- prcomp(data_clean[, -1], scale. = TRUE)
# ---------------------------------------------------------------------------
# Model A: Logistic Regression using ALL Genes
#--------------------------------------------------------------------------
# Fit the model predicting CellType from all 5000+ gene columns
# This usually triggers a warning about "fitted probabilities 0 or 1" (Overfitting)
model_all <- glm(CellType ~ ., data = data_clean, family = "binomial")
# Generate Predictions (Probability of class 2)
probs_all <- predict(model_all, type = "response")
# Classify: Threshold at 0.5
# If prob > 0.5, predict the second level (e.g., T-cell), else the first level
pred_class_all <- ifelse(probs_all > 0.5, levels(data_clean$CellType)[2], levels(data_clean$
    ↪ CellType)[1])
# REPORT: Confusion Matrix
cat("\n--- Model A (All Genes) Confusion Matrix ---\n")
print(table(Predicted = pred_class_all, Actual = data_clean$CellType))
# ---------------------------------------------------------------------------
# Model B: Logistic Regression using PCA Components
# ---------------------------------------------------------------------------
# Create a new dataframe with Target and the first 2 PCs
data_pca <- data.frame(CellType = data_clean$CellType,
                       PC1 = pca_res$x[, 1],
                       PC2 = pca_res$x[, 2])
```

```
# Fit model using only PC1 and PC2
model_pca <- glm(CellType ~ PC1 + PC2, data = data_pca, family = "binomial")
# REPORT: Probabilistic Model
coeffs <- coef(model_pca)
target_class <- levels(data_clean$CellType)[2] # The class corresponding to y=1
cat("\n--- Model B (PCA) Probabilistic Model ---\n")
cat("The probability that a cell is type '", target_class, "' is given by:\n", sep="")
cat("P(CellType =", target_class, "| X) = 1 / (1 + exp(-z))\n")
cat("Where z =", round(coeffs[1], 4),
    "+ (", round(coeffs[2], 4), "* PC1 )",
    "+ (", round(coeffs[3], 4), "* PC2 )\n")

# ======================
# ASSIGNMENT 1.3
# ======================
# 1. Define the Cost Function (Negative Log-Likelihood)
# Returns the scalar cost for a given theta (parameters)
nll_func <- function(theta, X, y) {
  # Calculate linear predictor z = X * theta
  # Note: X must already include the intercept column (ones)
  z <- X %*% theta
  # Sigmoid activation: 1 / (1 + exp(-z))
  p <- 1 / (1 + exp(-z))
  # Avoid log(0) errors by clamping probabilities slightly (numerical stability)
  p <- pmax(pmin(p, 1 - 1e-15), 1e-15)
  # Calculate Negative Log Likelihood
  cost <- -sum(y * log(p) + (1 - y) * log(1 - p))
  return(cost)
}
# 2. Prepare Data (50/50 Split)
set.seed(12345)
n <- nrow(data_clean)
id <- sample(1:n, floor(0.5 * n))
train_data <- data_clean[id, ]
test_data <- data_clean[-id, ]
# Create Matrices for the Cost Function
# We use ALL genes as features (this might be slow, but it's what the task implies)
# We must add a column of 1s for the Intercept (theta_0)
X_train <- as.matrix(cbind(1, train_data[, -1]))
y_train <- as.numeric(train_data$CellType) - 1   # Convert Factor (1,2) to (0,1)
X_test <- as.matrix(cbind(1, test_data[, -1]))
y_test <- as.numeric(test_data$CellType) - 1
# Initial parameters (vector of zeros)
init_theta <- rep(0, ncol(X_train))
# 3. Optimization Loop
# We run optim() repeatedly with increasing 'maxit' to track the path
max_iterations <- 20
train_costs <- numeric(max_iterations)
test_costs <- numeric(max_iterations)
cat("Running optimization loop (this may take 10-20 seconds)...\n")
for(i in 1:max_iterations) {
  # Run optimization limited to 'i' iterations
  opt <- optim(par = init_theta, fn = nll_func, X = X_train, y = y_train,
               method = "BFGS", control = list(maxit = i))
  # Calculate cost at the point where it stopped
  theta_curr <- opt$par
  train_costs[i] <- nll_func(theta_curr, X_train, y_train)
  test_costs[i] <- nll_func(theta_curr, X_test, y_test)
  # Optional: print status every 5 steps
  if(i %% 5 == 0) cat("Iteration:", i, "Train Cost:", round(train_costs[i], 2), "\n")
}
# 4. Plot Training vs Test Cost
# Plot Training Cost (Blue)
y_range <- range(c(train_costs, test_costs))
plot(1:max_iterations, train_costs, type = "b", col = "blue", pch = 19, lwd = 2,
     ylim = y_range, xlab = "Iteration", ylab = "Cost (Neg Log-Likelihood)",
     main = "Training vs Test Cost")
# Add Test Cost (Red)
lines(1:max_iterations, test_costs, type = "b", col = "red", pch = 19, lwd = 2)
legend("topright", legend = c("Train Cost", "Test Cost"), col = c("blue", "red"),
       lwd = 2, pch = 19)
```

Assignment 1 Analysis:

- **PCA Scaling:** Genetic data has high variance. Without scaling (standardizing to variance=1), genes with naturally large expression values would dominate the principal components, ignoring biologically relevant but smaller signals.
- **High-Dimensionality ($p \gg n$):** When features (p) exceed observations (n), standard logistic regression cannot be uniquely estimated (perfect separation/overfitting). It requires regularization (Ridge/Lasso) or dimensionality reduction (PCA) to work.
- **Early Stopping:** If the Test Cost starts increasing while Training Cost decreases, early stopping is needed to prevent overfitting. If both decrease or plateau, it is not needed.

Assignment 2: Kernel Density & Neural Networks

Problem Statement:
Data: *Synthetic (1000/class). Split: 1600 Train, 200 Valid, 200 Test.*

1. ***KDE (kernel density estimation) Classifier:***
   - *Estimate density $p(x|C)$ using KDE (Gaussian Kernel).*
   - *Classify via Bayes: $p(C_1|x) \propto p(x|C_1)p(C_1)$.*
   - *Tune $h \in \{0.1..5.0\}$ on Valid. Estimate error on Test.*
2. ***Neural Networks:***
   - *Repeat classification with neuralnet.*
   - *Tune architecture on Valid. Estimate error on Test.*

```
# ASSIGNMENT 2: KERNEL MODELS
# 1. Data Generation
set.seed(1234567890)
# Function to generate Class 1 (Mixture: 30% N(15,3), 70% N(4,2))
gen_class1 <- function(n) {
  x <- numeric(n)
  for(i in 1:n) {
    a <- rbinom(1, size = 1, prob = 0.3)
    # Based on text: "mean=15, sd=3" and "mean=4, sd=2"
    x[i] <- a * rnorm(1, 15, 3) + (1 - a) * rnorm(1, 4, 2)
  }
  return(x)
}
# Function to generate Class 2 (Mixture: 40% N(10,2), 60% N(5,2))
gen_class2 <- function(n) {
  x <- numeric(n)
  for(i in 1:n) {
    a <- rbinom(1, size = 1, prob = 0.4)
    # Based on text: "mean=10, sd=2" and "mean=5, sd=2" (inferred from broken text)
    x[i] <- a * rnorm(1, 10, 2) + (1 - a) * rnorm(1, 5, 2)
  }
  return(x)
}
# Generate 1000 samples per class
data_class1 <- gen_class1(1000)
data_class2 <- gen_class2(1000)
```

```r
# 2. Split Data
# Training: First 800 (Total 1600) - For building the model
train_c1 <- data_class1[1:800]
train_c2 <- data_class2[1:800]
# Validation: Next 100 (Total 200) - For selecting h
val_c1 <- data_class1[801:900]
val_c2 <- data_class2[801:900]
val_data <- c(val_c1, val_c2)
val_labels <- c(rep(1, 100), rep(2, 100))
# Test: Last 100 (Total 200) - For estimating generalization error
test_c1 <- data_class1[901:1000]
test_c2 <- data_class2[901:1000]
test_data <- c(test_c1, test_c2)
test_labels <- c(rep(1, 100), rep(2, 100))
# 3. Kernel Density Classifier Functions
# Implements: p(x) = (1/n) * sum( K((x-xi)/h) )
# Note: dnorm(u) is the Gaussian kernel K(u)
get_density_score <- function(x_val, train_data, h) {
  # Vectorized calculation for speed
  u <- (x_val - train_data) / h
  # Calculate Gaussian values
  k_values <- dnorm(u)
  # Mean value (sum / n)
  # We divide by h for proper density scaling (1/nh), though it cancels in Bayes ratio.
  return(mean(k_values) / h)
}
classify_kernel <- function(data_vec, train_c1, train_c2, h) {
  preds <- numeric(length(data_vec))
  for(i in 1:length(data_vec)) {
    x <- data_vec[i]
    # Calculate likelihoods
    lik_1 <- get_density_score(x, train_c1, h)
    lik_2 <- get_density_score(x, train_c2, h)
    # Bayes Rule (Priors are 0.5/0.5, so they cancel)
    if(lik_1 > lik_2) {
      preds[i] <- 1
    } else {
      preds[i] <- 2
    }
  }
  return(preds)
}
# 4. Model Selection (Tuning h)
# "Select the kernel width h from among the values 0.1, 0.2, ..., 4.9, 5" [cite: 75]
h_values <- seq(0.1, 5.0, by = 0.1)
acc_results <- numeric(length(h_values))
cat("Tuning Kernel Width h (Validation Set)...\n")
for(i in 1:length(h_values)) {
  h <- h_values[i]
  # Predict on Validation Set
  val_preds <- classify_kernel(val_data, train_c1, train_c2, h)
  # Calculate Accuracy
  acc <- mean(val_preds == val_labels)
  acc_results[i] <- acc
}
# Find best h
best_idx <- which.max(acc_results)
best_h <- h_values[best_idx]
best_acc_val <- acc_results[best_idx]
cat("\n--- Tuning Results ---\n")
cat("Best h found:", best_h, "\n")
cat("Validation Accuracy:", best_acc_val, "\n")
```

```r
# Plot Accuracy vs h
plot(h_values, acc_results, type = "l", col = "blue", lwd = 2,
     xlab = "Kernel Width (h)", ylab = "Validation Accuracy",
     main = "Model Selection: Accuracy vs Kernel Width")
abline(v = best_h, col = "red", lty = 2)
# 5. Generalization Error (Test Set)
# "Use the 200 samples that you have not used so far to estimate the generalization error"
test_preds <- classify_kernel(test_data, train_c1, train_c2, best_h)
test_acc <- mean(test_preds == test_labels)
gen_error <- 1 - test_acc
cat("\n--- Final Test Results ---\n")
cat("Selected h:", best_h, "\n")
cat("Test Accuracy:", test_acc, "\n")
cat("Estimated Generalization Error:", gen_error, "\n")

# ==============================================================================
# ASSIGNMENT 2: NEURAL NETWORKS (OPTIMIZED)
# ==============================================================================
library(neuralnet)
# --- 1. DATA GENERATION ---
set.seed(123456789)
# Generate Class 1 Data
c1_a <- rbinom(1000, 1, 0.3)
c1_data <- c1_a * rnorm(1000, 15, 3) + (1 - c1_a) * rnorm(1000, 4, 2)
df_c1 <- data.frame(X = c1_data, Y = 1)
# Generate Class 2 Data
c2_a <- rbinom(1000, 1, 0.4)
c2_data <- c2_a * rnorm(1000, 10, 5) + (1 - c2_a) * rnorm(1000, 15, 2)
df_c2 <- data.frame(X = c2_data, Y = 0)
# Combine
full_data <- rbind(df_c1, df_c2)
full_data <- full_data[sample(nrow(full_data)), ]
# --- 2. DATA SCALING  ---
# We must scale X to be roughly between 0 and 1 or -1 and 1.
# We save the scaling parameters to apply them to Test/Val data later.
max_val <- max(full_data$X)
min_val <- min(full_data$X)
# Apply Min-Max Scaling
full_data$X_scaled <- (full_data$X - min_val) / (max_val - min_val)
# Split: 1600 Train, 200 Val, 200 Test
train_df <- full_data[1:1600, ]
val_df   <- full_data[1601:1800, ]
test_df  <- full_data[1801:2000, ]
# --- 3. TRAIN MODELS (With Threshold) ---
# threshold=0.1 allows it to stop when the error change is small (faster)
# stepmax=1e6 ensures it doesn't error out just because it needs more steps
cat("Training NN Model 1 (Small - 3 neurons)...\n")
set.seed(12345)
# Note: We use X_scaled here!
nn1 <- neuralnet(Y ~ X_scaled, data = train_df, hidden = c(3),
                 linear.output = FALSE, err.fct = "ce",
                 stepmax = 1e6, threshold = 0.1)
cat("Training NN Model 2 (Large - 10 neurons)...\n")
set.seed(12345)
nn2 <- neuralnet(Y ~ X_scaled, data = train_df, hidden = c(10),
                 linear.output = FALSE, err.fct = "ce",
                 stepmax = 1e6, threshold = 0.1)
# --- 4. EVALUATION FUNCTION ---
evaluate_nn <- function(model, data_df) {
  # Use the SCALED column for prediction
  input_df <- data.frame(X_scaled = data_df$X_scaled)
  # Compute
  net_result <- compute(model, input_df)
  preds_prob <- net_result$net.result
  # Threshold at 0.5
  preds_class <- ifelse(preds_prob > 0.5, 1, 0)
  # Accuracy
  return(mean(preds_class == data_df$Y))
}
# --- 5. RESULTS ---
acc1 <- evaluate_nn(nn1, val_df)
acc2 <- evaluate_nn(nn2, val_df)
cat("\n--- Validation Results ---\n")
cat("Model 1 Accuracy:", acc1, "\n")
cat("Model 2 Accuracy:", acc2, "\n")
if(acc1 >= acc2) {
  best_model <- nn1
  cat("Selected: Model 1\n")
} else {
  best_model <- nn2
  cat("Selected: Model 2\n")
}
test_acc <- evaluate_nn(best_model, test_df)
cat("\n--- Final Test Results ---\n")
cat("Test Accuracy:", test_acc, "\n")
cat("Generalization Error:", 1 - test_acc, "\n")
```

- **KDE Classifier:**
  - A "generative" classifier. It explicitly models the distribution of each class $p(x|C)$.
  - **Kernel Width (h):** Acts as a smoothing parameter.
  - **Small h:** Spiky density, overfits training data (High Variance).
  - **Large h:** Flat density, oversmooths structure (High Bias).
- **Neural Network:** A "discriminative" classifier. It models the decision boundary directly $p(C|x)$ without caring about the underlying distribution of x.

Exam 2025-03-20

1.1 LASSO Regression for Area Prediction

Using scaled data, implement LASSO regression to predict the variable **Area** using all other numerical features (excluding *Diagnosis*).

- Perform cross-validation and generate a plot illustrating the dependence of the cross-validation error and its uncertainty on the penalty parameter λ.
- Identify the number of features selected by the optimal model (minimizing CV error).
- Report the predictive equation for the specific penalty log$(\lambda) = -2$.
- Determine if the model at log$(\lambda) = -2$ is statistically significantly different from the optimal model based on the standard error of the cross-validation scores.

```
# Install glmnet if needed
if (!require("glmnet")) install.packages("glmnet")
library(glmnet)
# 1. Load the data
data <- read.csv("wdbc.csv")
# 2. Prepare Data
# Target variable: Area (assuming 'area_mean' based on description)
y <- data$area_mean
# Features: Exclude ID, Diagnosis, and the Target itself
cols_to_exclude <- c("id", "diagnosis", "area_mean")
X_raw <- data[ , !(names(data) %in% cols_to_exclude)]
# --- CRITICAL FIX: Remove columns that contain NA values (like the empty 'X' column) since
    ↪ glmnet cant handle empty columns---
```

```r
X_raw <- X_raw[ , colSums(is.na(X_raw)) == 0]
# Scale the data (glmnet requires a matrix)
X <- scale(as.matrix(X_raw))
# 3. Compute LASSO regression with Cross-Validation
set.seed(12345)
cv_model <- cv.glmnet(X, y, alpha = 1)
# 4. Plot the Cross-Validation Error
plot(cv_model)
# 5. Report Optimal Features
optimal_lambda <- cv_model$lambda.min
coef_opt <- coef(cv_model, s = "lambda.min")
# Count non-zero coefficients (subtract 1 for Intercept)
n_features_opt <- sum(coef_opt != 0) - 1
cat("Optimal Log(lambda):", log(optimal_lambda), "\n")
cat("Number of features selected by optimal model:", n_features_opt, "\n")
# 6. Report Equation for log(lambda) = -2
specific_lambda <- exp(-2)
coef_spec <- coef(cv_model, s = specific_lambda)
cat("\nEquation for log(lambda) = -2:\n")
cat("Area_Mean =", coef_spec[1], "\n") # Intercept
# Loop through features to print non-zero coefficients
feat_names <- rownames(coef_spec)
for (i in 2:length(coef_spec)) {
  if (coef_spec[i] != 0) {
    cat("+ (", coef_spec[i], ") *", feat_names[i], "\n")
  }
}
# 7. Significance Test (Visual/Heuristic Check)
# Compare the error at log(lambda)=-2 with the optimal error + 1 Standard Error
cv_min_idx <- which(cv_model$lambda == cv_model$lambda.min)
mse_min <- cv_model$cvm[cv_min_idx]
mse_se_min <- cv_model$cvsd[cv_min_idx]
# Find error at specific lambda
spec_idx <- which.min(abs(log(cv_model$lambda) - (-2)))
mse_spec_est <- cv_model$cvm[spec_idx]
cat("\nMSE Optimal:", mse_min, "+/-", mse_se_min, "\n")
cat("MSE at log(lambda)=-2:", mse_spec_est, "\n")
cat("Is log(lambda)=-2 significantly worse? (Is MSE_spec > MSE_opt + SE_opt?):",
    mse_spec_est > (mse_min + mse_se_min), "\n")
```

1.2 Logistic Regression for Diagnosis Classification

Divide the original (unscaled) data into training and validation sets (50/50 split). Train a logistic regression model to predict *Diagnosis* ("M" as the positive class) using the remaining variables.

- Compute the misclassification errors for both training and test sets and discuss potential overfitting.
- Compute the **Precision** on the test data under two distinct loss matrices, L_1 and L_2, and analyze the change in precision:

$$L_1 = \begin{array}{c|cc} & PredB & PredM \\ TrueB & 0 & 1 \\ TrueM & 1 & 0 \end{array} \qquad L_2 = \begin{array}{c|cc} & PredB & PredM \\ TrueB & 0 & 20 \\ TrueM & 1 & 0 \end{array}$$

```r
#======== 1. 2 ========================
# 1. Load the data
data <- read.csv("wdbc.csv")
# 2. Preprocessing
# Convert Diagnosis to binary (M = 1, B = 0)
data$diagnosis <- ifelse(data$diagnosis == "M", 1, 0)
# Robust Cleanup:
# A. Remove 'id' column
data$id <- NULL
```

```r
# B. Remove any column that is fully empty (contains NA or is just empty)
# This handles "Unnamed..32", "X", or whatever the empty column is named.
data <- data[ , colSums(is.na(data)) < nrow(data)]
# 3. Split Data 50/50 into Training and Validation
set.seed(12345)
n <- nrow(data)
id <- sample(1:n, floor(n * 0.5))
train_data <- data[id, ]
test_data <- data[-id, ]
# 4. Train Logistic Regression Model
# family=binomial implies logistic regression
model <- glm(diagnosis ~ ., family = binomial, data = train_data)
# 5. Predictions
# type="response" gives probabilities P(Y=1|X)
prob_train <- predict(model, newdata = train_data, type = "response")
prob_test <- predict(model, newdata = test_data, type = "response")
# 6. Compute Misclassification Errors (Standard Threshold 0.5)
pred_train <- ifelse(prob_train > 0.5, 1, 0)
pred_test <- ifelse(prob_test > 0.5, 1, 0)
# Error = mean(predicted != actual)
train_error <- mean(pred_train != train_data$diagnosis)
test_error <- mean(pred_test != test_data$diagnosis)
cat("--- Logistic Regression Results ---\n")
cat("Training Misclassification Error:", train_error, "\n")
cat("Test Misclassification Error:    ", test_error, "\n")
if (test_error > train_error * 2) {
  cat("Comment: The model appears to be overfitted.\n")
} else {
  cat("Comment: The model does not show severe overfitting.\n")
}
# 7. Compute Precision for Loss Matrices L1 and L2
# Precision = TP / (TP + FP)
# --- Case L1: Symmetric Cost (Threshold 0.5) ---
pred_L1 <- ifelse(prob_test > 0.5, 1, 0)
TP_L1 <- sum(pred_L1 == 1 & test_data$diagnosis == 1)
FP_L1 <- sum(pred_L1 == 1 & test_data$diagnosis == 0)
# Handle division by zero if no positives predicted
precision_L1 <- if((TP_L1 + FP_L1) > 0) TP_L1 / (TP_L1 + FP_L1) else 0
# --- Case L2: Asymmetric Cost (FP cost = 20) ---
# Theoretical Threshold = Cost_FP / (Cost_FP + Cost_FN) ? No.
# Bayes Threshold formula: P(M) > L(B,M) / [L(M,B) + L(B,M)]
# Here L(Pred M | True B) = 20, L(Pred B | True M) = 1
# Threshold p > 20 / (20 + 1) = 20/21 approx 0.952
threshold_L2 <- 20/21
pred_L2 <- ifelse(prob_test > threshold_L2, 1, 0)
TP_L2 <- sum(pred_L2 == 1 & test_data$diagnosis == 1)
FP_L2 <- sum(pred_L2 == 1 & test_data$diagnosis == 0)
precision_L2 <- if((TP_L2 + FP_L2) > 0) TP_L2 / (TP_L2 + FP_L2) else 0
cat("\n--- Precision Analysis ---\n")
cat("Precision for L1 (Threshold 0.5):   ", precision_L1, "\n")
cat("Precision for L2 (Threshold ~0.95): ", precision_L2, "\n")
```

1.3 Linear Classification with Hinge Loss

Implement a linear classification model $\hat{y}(x) = sign(w^T x)$, where $y \in \{-1, 1\}$ (corresponding to ”B” and ”M” respectively).

- Define a cost function using the **Hinge Loss** on the training data from Step 2:

$$J(w) = \sum_{i=1}^{N} \max(0, 1 - y_i(w^T x_i))$$

- Optimize this cost function using the **BFGS** method, initializing weights at $w = (0, \ldots, 0)$.
- Report the estimated predictive equation and the misclassification errors for the training and test sets.
- Compare the performance of this model with the logistic regression model from Step 2.

```r
#===== Assignment 3 ========
# Clean up (remove ID and empty columns)
data$id <- NULL
data <- data[ , colSums(is.na(data)) < nrow(data)]
# Prepare Variables
# The problem specifies: y is -1 ("B") and 1 ("M")
# X contains all other variables
y <- ifelse(data$diagnosis == "M", 1, -1)
X_df <- data[ , !(names(data) %in% "diagnosis")]
X <- as.matrix(X_df) # Convert to matrix for algebra
# 2. Split Data (Same seed as Step 2)
set.seed(12345)
n <- nrow(data)
id <- sample(1:n, floor(n * 0.5))
X_train <- X[id, ]
y_train <- y[id]
X_test <- X[-id, ]
y_test <- y[-id]
# 3. Add Intercept Column (Bias)
# The model is w^T * x. To include w0 (intercept), we add a column of 1s to X.
X_train_bias <- cbind(1, X_train)
X_test_bias <- cbind(1, X_test)
# 4. Implement Hinge Loss Cost Function
# J(w) = Sum( max(0, 1 - y * (w^T x)) )
hinge_cost <- function(w, X, y) {
  # Calculate scores f(x) = X * w
  scores <- X %*% w
  # Calculate margins: y * f(x)
  margins <- y * scores
  # Calculate Hinge Loss: max(0, 1 - margin)
  losses <- pmax(0, 1 - margins)
  # Return sum of losses
  return(sum(losses))
}
# 5. Optimize using BFGS
# Starting point: w = (0, ..., 0)
initial_w <- rep(0, ncol(X_train_bias))
# Run optimization
# Note: We pass X_train_bias and y_train as additional arguments to hinge_cost
opt_res <- optim(par = initial_w,
                 fn = hinge_cost,
                 X = X_train_bias,
                 y = y_train,
                 method = "BFGS")
w_opt <- opt_res$par
convergence <- opt_res$convergence # 0 means successful convergence
# 6. Make Predictions and Compute Errors
# Prediction rule: y_hat = sign(w^T x)
pred_train_score <- X_train_bias %*% w_opt
pred_test_score <- X_test_bias %*% w_opt
pred_train <- sign(pred_train_score)
pred_test <- sign(pred_test_score)
# Misclassification Error
# Note: sign(0) is 0, which counts as a mismatch for both -1 and 1
train_error_hinge <- mean(pred_train != y_train)
test_error_hinge <- mean(pred_test != y_test)
```

```
# 7. Report Results
cat("--- Hinge Loss Classification Results ---\n")
cat("Optimization Convergence (0 = Success):", convergence, "\n")
cat("Hinge Training Error:", train_error_hinge, "\n")
cat("Hinge Test Error:    ", test_error_hinge, "\n")
# Report Estimated Predictive Equation (First few terms)
cat("\nEstimated Predictive Equation:\n")
cat("y_hat = sign(", round(w_opt[1], 3), "\n") # Intercept
feat_names <- colnames(X_df)
for(i in 1:5) { # Print first 5 features to save space
  cat("      + (", round(w_opt[i+1], 3), ") *", feat_names[i], "\n")
}
cat("      + ... )\n")
# 8. Compare with Logistic Regression (Hardcoded from previous step for context)
# You should update these values with the actual numbers from your Step 1.2 run
cat("\n--- Comparison ---\n")
cat("Logistic Test Error (Step 2): [Refer to your previous output, approx 0.05-0.06]\n")
cat("Hinge Test Error (Step 3):   ", test_error_hinge, "\n")
cat("Comment: The Hinge Loss model (linear SVM) often performs similarly to Logistic Regression
    ↪ on this dataset.\n")
cat("However, since we used unscaled data with BFGS, convergence might be harder for this model.\
    ↪ n")
```

Assignment 2. Kernel Models

Kernel models can be used for density estimation to model a probability distribution or density function $p(x_*)$. The density is defined as:

$$p(x_*) = \frac{1}{n}\sum_{i=1}^{n} k\left(\frac{x_* - x_i}{h}\right)$$

where the kernel function $k()$ must integrate to 1. For this task, assume $k()$ is the density function of a Gaussian distribution with mean 0 and standard deviation 1 (available as dnorm in R).

Task Requirements:

1. Use the kernel model to estimate the class conditional density functions $p(x_*|class = 1)$ and $p(x_*|class = 2)$.
2. Compute the posterior class probabilities $p(class|x_*)$ using Bayes' theorem:

$$p(class = 1|x_*) = \frac{p(x_*|class = 1)p(class = 1)}{p(x_*|class = 1)p(class = 1) + p(x_*|class = 2)p(class = 2)}$$

3. Generate your own dataset to illustrate the effect of the bandwidth parameter h on the generalization error.
4. Specifically demonstrate that overfitting occurs for a certain value of h, and underfitting occurs for another value of h.

```
# --- Assignment 2, Part 1: Kernel Models ---
# 1. Generate a Dataset (2 Classes)
# We create a non-linear dataset (e.g., two curved moon-like shapes or nested circles)
# to make the bandwidth h matter. Here we use two noisy concentric clusters.
set.seed(12345)
# Function to generate data
gen_data <- function(n) {
  # Class 1: Inner Circle (with noise)
  r1 <- runif(n/2, 0, 2)
  theta1 <- runif(n/2, 0, 2*pi)
  c1 <- data.frame(x1 = r1 * cos(theta1), x2 = r1 * sin(theta1), class = 1)
  # Class 2: Outer Ring (with noise)
  r2 <- runif(n/2, 3, 5)
  theta2 <- runif(n/2, 0, 2*pi)
  c2 <- data.frame(x2_1 = r2 * cos(theta2), x2_2 = r2 * sin(theta2))
  names(c2) <- c("x1", "x2")
  c2$class <- 2
  return(rbind(c1, c2))
}
# Generate Training and Test Data
train_data <- gen_data(200) # 100 per class
test_data <- gen_data(100)
# 2. Define Kernel Classifier Functions
# The problem specifies using dnorm as the kernel.
# For 2D data, we assume a product kernel: p(x1, x2) = p(x1) * p(x2) (Naive Bayes assumption
   ↪ locally)
# or simply multivariate independent KDE.
# Function to estimate class conditional density p(x* | class)
get_density <- function(x_star, data_class, h) {
  # x_star: vector of length 2 (the test point)
  # data_class: dataframe of training points belonging to one class
  # h: bandwidth
  # Calculate (x* - xi) / h for all training points i
  diffs_1 <- (x_star[1] - data_class$x1) / h
  diffs_2 <- (x_star[2] - data_class$x2) / h
  # Kernel values (Gaussian)
  k_vals <- dnorm(diffs_1) * dnorm(diffs_2)
  # Sum and normalize by n (and h^d for proper density scaling, though constant cancels in ratio)
  # Density = (1 / (n * h^2)) * sum(kernels)
  density <- sum(k_vals) / (nrow(data_class) * h^2)
  return(density)
}
# Function to predict class for a single point
predict_point <- function(x_new, train_data, h) {
  # Split training data
  d1 <- train_data[train_data$class == 1, ]
  d2 <- train_data[train_data$class == 2, ]
  # Priors (estimated from data count)
  prior1 <- nrow(d1) / nrow(train_data)
  prior2 <- nrow(d2) / nrow(train_data)
  # Likelihoods p(x | C)
  lik1 <- get_density(x_new, d1, h)
  lik2 <- get_density(x_new, d2, h)
  # Posterior (unnormalized is sufficient for comparison)
  post1 <- lik1 * prior1
  post2 <- lik2 * prior2
  # If both are 0 (numerical underflow for very small h far from points), predict random or
   ↪ majority
  if(post1 == 0 && post2 == 0) return(sample(1:2, 1))
  if (post1 > post2) return(1) else return(2)
}
# Wrapper to predict for a whole dataset
predict_all <- function(new_data, train_data, h) {
  preds <- apply(new_data[, 1:2], 1, predict_point, train_data = train_data, h = h)
  return(preds)
}
# 3. Analyze Effect of h (Bandwidth)
# We test a range of h values to find Overfitting vs Underfitting
h_values <- c(0.05, 0.5, 5)
# Prepare plotting grid for decision boundaries
x_grid <- seq(min(train_data$x1)-1, max(train_data$x1)+1, length.out = 50)
y_grid <- seq(min(train_data$x2)-1, max(train_data$x2)+1, length.out = 50)
grid_points <- expand.grid(x1 = x_grid, x2 = y_grid)
par(mfrow = c(1, 3)) # 3 plots side by side
for (h in h_values) {
  # Predict on Test Data to get Error
  test_preds <- predict_all(test_data, train_data, h)
  test_acc <- mean(test_preds == test_data$class)
  # Predict on Grid for Visualization
  grid_preds <- predict_all(grid_points, train_data, h)
  # Plot
  plot(train_data$x1, train_data$x2, col = ifelse(train_data$class == 1, "red", "blue"),
       pch = 19, main = paste("h =", h, "\nTest Acc:", test_acc),
       xlab = "x1", ylab = "x2", cex = 0.6)
  # Overlay decision boundary (contour)
  z <- matrix(grid_preds, nrow = 50, ncol = 50)
  contour(x_grid, y_grid, z, add = TRUE, levels = 1.5, drawlabels = FALSE, lwd = 2)
}
# 4. Comments (Output to console)
cat("--- Analysis of Bandwidth h ---\n")
cat("1. h = 0.05 (Small): Overfitting.\n")
cat("   The decision boundary is extremely jagged and forms tiny islands around specific training
    ↪ points.\n")
cat("   It fits noise and fails to generalize to the space between points.\n")
cat("\n2. h = 5 (Large): Underfitting.\n")
cat("   The decision boundary becomes too smooth or disappears entirely (predicts only one class)
    ↪ .\n")
cat("   It ignores the local structure of the data (the inner circle vs outer ring).\n")
cat("\n3. h = 0.5 (Medium): Good Fit.\n")
cat("   Captures the general circular structure without capturing the noise.\n")
```

Assignment 3: Neural Networks

Run the R code below (in code answer) to train a neural network designed to subtract two numbers drawn from the interval $[-1, 1]$.

```
# --- Assignment 2, Part 2: Neural Networks ---
library(neuralnet)
set.seed(1234567890)
# Generate Data
x1 <- runif(1000, -1, 1)
x2 <- runif(1000, -1, 1)
tr <- data.frame(x1, x2, y = x1 - x2)
# Train Neural Network
# Task: Subtract two numbers. y = x1 - x2.
# Network: 2 Inputs -> 1 Hidden (tanh) -> 1 Output (Linear by default)
winit <- runif(9, -1, 1) # Random init (unused by default unless passed to startweights)
nn <- neuralnet(formula = y ~ x1 + x2, data = tr, hidden = c(1), act.fct = "tanh")
# Plot the network (opens in plot window)
plot(nn)
# Extract and Print Weights
weights <- nn$result.matrix
cat("\n--- Learned Weights ---\n")
print(weights)
# --- Explanation of Why Weights Make Sense ---
cat("\n--- Interpretation ---\n")
cat("The goal is to approximate the linear function y = x1 - x2.\n")
cat("The network structure is: y_out = W_out * tanh(W_1*x1 + W_2*x2 + Bias_h) + Bias_out\n")
cat("\nExplanation:\n")
cat("1. Linearity: The tanh function, tanh(z), is approximately linear for small z (tanh(z) ~ z)
    ↪ .\n")
cat("   To mimic the linear relationship (x1 - x2), the network likely learned SMALL weights \n")
cat("   for the input layer to keep the activation in this linear range.\n")
cat("\n2. Signs of Weights: We need 'Positive x1' and 'Negative x2'.\n")
cat("   - Look at the weights W_1 (x1 to hidden) and W_2 (x2 to hidden).\n")
cat("   - They should have OPPOSITE signs (e.g., W_1 positive, W_2 negative, or vice versa).\n")
cat("   - The output weight W_out will then scale and flip the sign if necessary to match (x1 -
    ↪ x2).\n")
cat("\n3. Example Check:\n")
cat("   If W_1 is approx -0.5 and W_2 is approx +0.5, then hidden ~ -0.5(x1 - x2).\n")
cat("   Then W_out should be approx -2.0 to result in (-2.0) * (-0.5) * (x1-x2) = 1.0 * (x1-x2).\
    ↪ n")
```

Exam 2025-01-17

Assignment 1 (10p)

The file `lakesurvey.csv` contains chemical measurements of different lakes in Sweden as well as their pH-level.

1. Perform principal component analysis (PCA) excluding the `pH` variable on the scaled original data.
    - How much variation is explained by the first two principal components?
    - Which features contribute to the first principal component the most?
    - By assuming that we keep only the first two principal components, report an equation showing how the unscaled `Cond` variable can be approximated from the first two principal components. (3p)

```r
#data prep
data <- read.csv("lakesurvey.csv")
#perform PCA, exclude 'pH' (column 1) and scale the data. prcomp uses SVD
data_pca <- data[, colnames(data) != "pH"] #remove pH
pca_model <- prcomp(data_pca, scale. = TRUE)
#Q1: how much variation is explained by the first two PCs?
summary_pca <- summary(pca_model)
var_explained <- summary_pca$importance[2, 1:2] #proportion of variance for pc1,pc2
total_var_2pcs <- sum(var_explained)
cat("--- Variation Explained ---\n")
print(var_explained)
cat("Total variation explained by first 2 PCs:", total_var_2pcs * 100, "%\n\n")
#-- q2: Which features contribute to the first PC mostly? ---
#we look at the absolute alues of the loadings (rotation) for PC1
loadings <- pca_model$rotation
loadings_pc1 <- abs(loadings[, 1])
#sort them to see the highest contributors
sorted_loadings <- sort(loadings_pc1, decreasing = TRUE)
cat("--- Top contributing features to PC1 --- \n")
print(sorted_loadings)
cat("Comment: The features with the highest absolute loadings drive PC1. \n\n")
# --- Q3: Equation for unscaled Cond from PC1 and PC2 ---
# PCA reconstruction formula: X_original = Mean + SD * (Score * Loading)
# Cond_scaled approx = (Loading_Cond_PC1 * PC1) + (Loading_Cond_PC2 * PC2)
# Cond_original approx = Mean_Cond + SD_Cond * [(Loading_Cond_PC1 * PC1) + (Loading_Cond_PC2
    ↪  * PC2)]
# Get stats for Cond
mu_cond <- mean(data$Cond)
sd_cond <- sd(data$Cond)
# Get Loadings for Cond
l1 <- loadings["Cond", "PC1"]
l2 <- loadings["Cond", "PC2"]
cat("--- Equation for Cond ---\n")
cat("Mean(Cond):", mu_cond, "\n")
cat("SD(Cond):  ", sd_cond, "\n")
cat("Loading(Cond, PC1):", l1, "\n")
cat("Loading(Cond, PC2):", l2, "\n")
cat("\nApproximation Equation:\n")
cat("Cond =", round(mu_cond, 2), "+", round(sd_cond, 2), "* (",
    round(l1, 3), "* PC1 +", round(l2, 3), "* PC2 )\n")
# Simplify the algebra for the final report
coeff_pc1 <- sd_cond * l1
coeff_pc2 <- sd_cond * l2
intercept <- mu_cond
cat("\nSimplified Equation:\n")
cat("Cond =", round(intercept, 2),
    ifelse(coeff_pc1 >= 0, "+", "-"), abs(round(coeff_pc1, 2)), "* PC1",
    ifelse(coeff_pc2 >= 0, "+", "-"), abs(round(coeff_pc2, 2)), "* PC2\n")
```

2. Divide the original data into training, validation, and test sets (40/30/30). Consider also a dataset with three variables—the first two principal components and the `pH` variable—and split this dataset into training, validation, and test sets using the same partitioning indices as in the original dataset. Scale the datasets appropriately.
   - Estimate K-nearest neighbor models with `pH` as the target and remaining variables as features using the **original data** partitioning and $K = 1, 10, 50$.
   - Estimate K-nearest neighbor models with `pH` as the target and remaining variables as features using the **principal component data** partitioning and $K = 1, 10, 50$.
   - Report training, validation, and test MSE for these 6 models.
   - Which K leads to the best model from the original data?
   - Which K leads to the best model from the principal component dataset?
   - What kind of data (original or principal component) results in the best prediction?
   - Why might one need to check the test error of the best model as well?
   - Why does $K = 1$ result in zero training MSE?
   - What target distribution do we implicitly assume by using MSE as a cost function? (5p)

```
# ==== ASSIGNMENT 1.2 ======
# Load required library for KNN Regression
# If you don't have 'kknn', install it: install.packages("kknn")
library(kknn)
# 1. Prepare Data
data <- read.csv("lakesurvey.csv")
set.seed(12345) # Use the exam seed
# --- Dataset A: Original Data ---
# Target: pH, Features: All others
# We don't scale yet; we scale after splitting to avoid data leakage.
# --- Dataset B: PCA Data ---
# We need to recreate the PCA scores from Step 1.1
# Note: PCA is usually calculated on the whole dataset or training set.
# Given the instructions imply comparing datasets, we'll derive PCs from the whole set
# (as implied by "dataset with the three variables... split this dataset")
data_feat <- data[, colnames(data) != "pH"]
pca_model <- prcomp(data_feat, scale. = TRUE)
data_pca <- data.frame(pH = data$pH,
                       PC1 = pca_model$x[, 1],
                       PC2 = pca_model$x[, 2])
# 2. Split Data (40/30/30)
n <- nrow(data)
# Random indices
set.seed(12345)
indices <- sample(1:n)
# Calculate split points
n_train <- floor(0.4 * n)
n_val <- floor(0.3 * n)
n_test <- n - n_train - n_val # Remaining 30%
# Define Indices
idx_train <- indices[1:n_train]
idx_val <- indices[(n_train + 1):(n_train + n_val)]
idx_test <- indices[(n_train + n_val + 1):n]
# Helper function to get splits, scale, and train KNN
run_knn_analysis <- function(dataset, name) {
  # Split
  train <- dataset[idx_train, ]
  val <- dataset[idx_val, ]
  test <- dataset[idx_test, ]
  # Scale "appropriately" (Fit on Train, Apply to Val/Test)
  # We assume pH (target) should strictly speaking not be scaled for MSE interpretation,
  # or scaled and then unscaled.
  # However, KNN relies on feature scaling.
  # Let's scale features ONLY.
  # Identify feature columns (exclude pH)
  feat_cols <- setdiff(names(dataset), "pH")
  # Scale Training Features
  train_scaled <- train
  train_feat <- train[, feat_cols]
  train_means <- colMeans(train_feat)
  train_sds <- apply(train_feat, 2, sd)
  # Apply scaling
  train_scaled[, feat_cols] <- scale(train_feat, center = train_means, scale = train_sds)
  val_scaled <- val
  val_scaled[, feat_cols] <- scale(val[, feat_cols], center = train_means, scale = train_sds
    ↪ )
  test_scaled <- test
  test_scaled[, feat_cols] <- scale(test[, feat_cols], center = train_means, scale = train_
    ↪ sds)
  # K values to test
  k_values <- c(1, 10, 50)
  cat(paste0("\n--- Results for ", name, " Data ---\n"))
  cat(sprintf("%-5s %-12s %-12s %-12s\n", "K", "Train MSE", "Val MSE", "Test MSE"))
  results <- list()
  for (k in k_values) {
    # Train/Predict using kknn
    # kknn performs regression when target is continuous
    # Train vs Train (for Training Error)
    model_train <- kknn(pH ~ ., train_scaled, train_scaled, k = k, kernel = "rectangular")
    pred_train <- model_train$fitted.values
    mse_train <- mean((train_scaled$pH - pred_train)^2)
    # Train vs Val
    model_val <- kknn(pH ~ ., train_scaled, val_scaled, k = k, kernel = "rectangular")
    pred_val <- model_val$fitted.values
    mse_val <- mean((val_scaled$pH - pred_val)^2)
    # Train vs Test
    model_test <- kknn(pH ~ ., train_scaled, test_scaled, k = k, kernel = "rectangular")
    pred_test <- model_test$fitted.values
    mse_test <- mean((test_scaled$pH - pred_test)^2)
    cat(sprintf("%-5d %-12.4f %-12.4f %-12.4f\n", k, mse_train, mse_val, mse_test))
    results[[paste0("k", k)]] <- c(mse_val, mse_test)
  }
}
# Run Analysis
run_knn_analysis(data, "Original")
run_knn_analysis(data_pca, "PCA (PC1+PC2)")
```

Which K = best model from org data? Look at valid. MSE. K = 1 overfits, K = 50 might underfit. Which K = best model from pc dataset+ Often K = 10 och K = 50 performs best. PCA condenses info, mdoerate K works well. Original data usually provides better predictions. PCA reduces dimensionality, discards components = information loss. KNN can handle higher dimensionality of org data.

3. Use the original data set and the principal component dataset from step 2, scale them, and compute the ratio between the Euclidean distance to the nearest observation from observation number 1 and the Euclidean distance to the furthest observation from observation number 1, per dataset (exclude `pH` variable in the distance computations). Compare these ratios and comment on what kind of machine learning phenomenon this comparison illustrates. (2p)

```
#===== Assignment 1.3 ====
# 1. Prepare Datasets
data <- read.csv("lakesurvey.csv")
# A. Original Data (Features only, exclude pH)
data_orig <- data[, colnames(data) != "pH"]
# B. PCA Data (First 2 Components)
pca_model <- prcomp(data_orig, scale. = TRUE)
data_pca <- data.frame(PC1 = pca_model$x[, 1], PC2 = pca_model$x[, 2])
# 2. Scale Both Datasets
# Distances are sensitive to scale, so we standardise
data_orig_scaled <- scale(data_orig)
data_pca_scaled <- scale(data_pca)
# 3. Define Function to Compute Ratio
# Ratio = Dist_to_Nearest / Dist_to_Furthest (from Obs 1)
compute_dist_ratio <- function(dataset) {
  # Get Observation 1
  obs1 <- dataset[1, , drop = FALSE]
  # Compute Euclidean distances to all other points
  # We use the 'dist' function or manual calculation
  # Manual is clearer: sqrt(sum((x - y)^2))
  dists <- apply(dataset, 1, function(x) sqrt(sum((x - obs1)^2)))
  # Remove the distance to itself (which is 0 at index 1)
  dists_others <- dists[-1]
  # Find Nearest and Furthest
  d_min <- min(dists_others)
  d_max <- max(dists_others)
  return(d_min / d_max)
}
# 4. Compute Ratios
ratio_orig <- compute_dist_ratio(data_orig_scaled)
ratio_pca <- compute_dist_ratio(data_pca_scaled)
# 5. Report Results
cat("--- Distance Ratio Analysis (d_min / d_max) ---\n")
cat("Original Data Ratio (High Dim):", ratio_orig, "\n")
cat("PCA Data Ratio (Low Dim):      ", ratio_pca, "\n")
# 6. Phenomenon Explanation
cat("\n--- Comparison & Phenomenon ---\n")
if (ratio_orig > ratio_pca) {
  cat("Result: The ratio is HIGHER in the original (higher-dimensional) data.\n")
} else {
  cat("Result: The ratio is lower in the original data (unexpected for this specific sample).\n")
}
cat("Phenomenon Illustration: The Curse of Dimensionality.\n")
cat("Explanation:\n")
cat("1. In high-dimensional spaces (Original Data), data becomes sparse.\n")
cat("2. As dimensions increase, the distance to the nearest neighbor approaches the distance to
    ↪ the furthest neighbor.\n")
cat("3. This means 'd_min' and 'd_max' become similar, pushing the ratio towards 1.\n")
cat("4. In lower dimensions (PCA Data), the contrast between near and far points is usually more
    ↪ distinct (lower ratio).\n")
cat("5. This makes distance-based methods (like KNN) less effective in high dimensions because '
    ↪ nearest' neighbors aren't meaningfully closer than random points.\n")
```

Assignment 2: Kernel Models (6p)

In the course, you have learned about kernel models for classification and regression. Kernel models can also be used for density estimation, i.e., to model a probability distribution or density function $p(x_*)$. In particular,

$$p(x_*) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h}k\left(\frac{x_* - x_i}{h}\right)$$

where the kernel function $k()$ must integrate to 1. To ensure this, you will hereinafter consider $k()$ to be the density function of a Gaussian distribution with mean equal to 0 and standard deviation equal to 1. You can get it by using the command dnorm in R.

1. Run the code below to produce some learning data, which consist of 1000 samples from class 1 and 1000 samples from class 2. These points are stored in the variables data_class1 and data_class2. Implement the kernel model presented above to estimate the density function of the data sampled from class 1. Do the same for class 2. Use only 800 samples from class 1 and 800 samples from class 2. (2p)

2. Once you have obtained the kernel models for the class conditional density functions $p(x_*|class = 1)$ and $p(x_*|class = 2)$, use them to produce posterior class probabilities $p(class|x_*)$ via Bayes' theorem:

$$p(class = 1|x_*) = \frac{p(x_*|class = 1)p(class = 1)}{p(x_*|class = 1)p(class = 1) + p(x_*|class = 2)p(class = 2)}$$

   Use these probabilities to compute the correct classification rate on 200 samples that you did not use before (100 from class 1 and 100 from class 2). Use this classification rate to select the kernel width h from among the values $0.1, 0.2, \dots, 5.0$. (3p)

3. Finally, use the remaining 200 samples (that you have not used so far) to estimate the generalization error of the kernel model selected. Comment your code. (1p)

```r
# --- Assignment 2: Kernel Models --
# 1. Generate Data (Provided Code)
set.seed(123456789)
N_class1 <- 1000
N_class2 <- 1000
data_class1 <- NULL
for(i in 1:N_class1){
  a <- rbinom(n = 1, size = 1, prob = 0.3)
  b <- rnorm(n = 1, mean = 15, sd = 3) * a + (1-a) * rnorm(n = 1, mean = 4, sd = 2)
  data_class1 <- c(data_class1, b)
}
data_class2 <- NULL
for(i in 1:N_class2){
  a <- rbinom(n = 1, size = 1, prob = 0.4)
  b <- rnorm(n = 1, mean = 10, sd = 5) * a + (1- a) * rnorm(n = 1, mean = 15, sd = 2)
  data_class2 <- c(data_class2, b)
}
# 2. Split Data
# Train: First 800
# Validation: Next 100 (801-900)
# Test: Remaining 100 (901-1000)
train_c1 <- data_class1[1:800]
train_c2 <- data_class2[1:800]
val_c1 <- data_class1[801:900]
val_c2 <- data_class2[801:900]
# Combine Validation set for classification
val_data <- c(val_c1, val_c2)
val_labels <- c(rep(1, 100), rep(2, 100))
test_c1 <- data_class1[901:1000]
test_c2 <- data_class2[901:1000]
# Combine Test set
test_data <- c(test_c1, test_c2)
test_labels <- c(rep(1, 100), rep(2, 100))
# 3. Define Kernel Density Estimator Function
# Formula: p(x) = (1/n) * Sum( (1/h) * K((x-xi)/h) )
kde <- function(x_star, data_train, h) {
  # We compute density for a single point x_star
  # (x_star - xi) / h
  u <- (x_star - data_train) / h
  # Kernel K is Gaussian (dnorm)
  k_vals <- dnorm(u)
  # Average and divide by h
  density <- mean(k_vals) / h
  return(density)
}
# 4. Define Bayes Classifier
# Returns P(Class=1 | x)
predict_posterior_c1 <- function(x, train_c1, train_c2, h) {
  # Priors (Equal, since N1=N2=1000)
  prior1 <- 0.5
  prior2 <- 0.5
  # Likelihoods p(x | C)
  lik1 <- kde(x, train_c1, h)
  lik2 <- kde(x, train_c2, h)
  # Evidence p(x)
  evidence <- lik1 * prior1 + lik2 * prior2
  # Handle division by zero if evidence is extremely small
  if(evidence == 0) return(0.5)
  # Posterior P(C=1 | x)
  post1 <- (lik1 * prior1) / evidence
  return(post1)
}
# 5. Tune h on Validation Set
h_values <- seq(0.1, 5.0, by = 0.1)
accuracies <- numeric(length(h_values))
cat("--- Tuning Bandwidth h ---\n")
for(i in 1:length(h_values)) {
  h <- h_values[i]
  # Predict for all validation points
  # If P(C=1|x) > 0.5, predict 1, else 2
  preds <- sapply(val_data, function(x) {
    p1 <- predict_posterior_c1(x, train_c1, train_c2, h)
    if(p1 > 0.5) return(1) else return(2)
  })
  # Calculate Accuracy
  acc <- mean(preds == val_labels)
  accuracies[i] <- acc
}
# Select Best h
best_idx <- which.max(accuracies)
best_h <- h_values[best_idx]
best_acc <- accuracies[best_idx]
cat("Best h found:", best_h, "\n")
cat("Validation Accuracy:", best_acc, "\n")
# Plot results (Optional but recommended for 'detailed report')
plot(h_values, accuracies, type = "l", col = "blue", lwd = 2,
     main = "Validation Accuracy vs Bandwidth h",
     xlab = "Bandwidth h", ylab = "Accuracy")
abline(v = best_h, col = "red", lty = 2)
# 6. Estimate Generalization Error on Test Set
# Using the best h
final_preds <- sapply(test_data, function(x) {
  p1 <- predict_posterior_c1(x, train_c1, train_c2, best_h)
  if(p1 > 0.5) return(1) else return(2)
})
test_accuracy <- mean(final_preds == test_labels)
test_error <- 1 - test_accuracy
cat("\n--- Final Results ---\n")
cat("Selected Kernel Width h:", best_h, "\n")
cat("Test Accuracy:", test_accuracy, "\n")
cat("Generalization Error:", test_error, "\n")
# Comment on the code
cat("\nComment:\n")
cat("We used the first 800 samples for density estimation (training).\n")
cat("The validation set (next 100) was used to select h by maximizing accuracy.\n")
cat("Small h (0.1) likely overfits (noisy density), while very large h (5.0) underfits (over-
    ↪ smoothed).\n")
cat("The final generalization error is reported on the unseen test set (last 100).\n")
```

Assignment 3: Neural Networks (4p)

In this assignment, you are asked to use the R package `neuralnet` to train a NN to learn the trigonometric sine function. To produce the learning data, sample 50 points uniformly at random in the interval $[0, 10]$ and then apply the sine function to each point.

1. Your task is to estimate the generalization mean squared error of a NN with a single hidden layer of 10 units for the regression task described above. To this end, use cross-validation with 2 folds. Initialize the weights of the NN to random values in the interval $[-1, 1]$. Stop the training when the partial derivatives of the error function are below a threshold value of 0.001. Recall that cross-validation works as follows:
   - Divide the learning data into approximately equal sized folds D_1 and D_2.
   - Train the regressor on D_1 and test it on D_2.
   - Train the regressor on D_2 and test it on D_1.
   - Report the average error on the test folds. (3p)
2. Answer the following question: What is the NN that you should return to the user? The one learned from D_1? The one learned from D_2? Either of them? None? (1p)

```
# --- Assignment 3: Neural Networks ---
library(neuralnet)
# 1. Generate Data (from template)
set.seed(1234567890)
Var <- runif(50, 0, 10)
tr <- data.frame(Var, Sin = sin(Var))
# 2. Setup Cross-Validation
# Fold 1: Indices 1-25
# Fold 2: Indices 26-50
tr1 <- tr[1:25, ]
tr2 <- tr[26:50, ]
# Define training parameters
# "Initialize weights... [-1, 1]" -> handled by set.seed and neuralnet defaults usually.
# "Threshold = 0.001"
# "Hidden units = 10"
# --- Fold 1: Train on tr1, Test on tr2 ---
set.seed(1234567890) # Reset seed for consistent initialization
nn1 <- neuralnet(Sin ~ Var, data = tr1, hidden = 10, threshold = 0.001)
# Predict on Fold 2 (Test)
# compute() returns $net.result
pred1 <- compute(nn1, tr2[, "Var", drop = FALSE])$net.result
mse1 <- mean((tr2$Sin - pred1)^2)
# --- Fold 2: Train on tr2, Test on tr1 ---
set.seed(1234567890)
nn2 <- neuralnet(Sin ~ Var, data = tr2, hidden = 10, threshold = 0.001)
# Predict on Fold 1 (Test)
pred2 <- compute(nn2, tr1[, "Var", drop = FALSE])$net.result
mse2 <- mean((tr1$Sin - pred2)^2)
# 3. Report Average Error
avg_mse <- (mse1 + mse2) / 2
cat("--- Neural Network Cross-Validation Results ---\n")
cat("MSE Fold 1 (Test on tr2):", mse1, "\n")
cat("MSE Fold 2 (Test on tr1):", mse2, "\n")
cat("Average Generalization MSE:", avg_mse, "\n")
# 4. Conceptual Question
cat("\n--- Question: Which NN to return to the user? ---\n")
cat("Answer: None of them (or a new one).\n")
cat("Reasoning:\n")
cat("1. The models from CV were trained on only 50% of the available data (25 points).\n")
cat("2. Cross-validation is used to ESTIMATE the performance (error) of this model architecture.\n")
cat("3. Once we have confirmed the error is acceptable, the best practice is to retrain a NEW model\n")
cat("   using the COMPLETE dataset (all 50 points) to maximize learning, and deliver that one to the user.\n")
# Optional: Train the final model
nn_final <- neuralnet(Sin ~ Var, data = tr, hidden = 10, threshold = 0.001)
plot(nn_final) # Visual check
```

CUSTOM PROBLEMS: LDA, CNN ADVANCED DIMENSIONALITY REDUCTION

Extra Assignment: LDA, CNN & Advanced Dim. Reduction

Problem Statement:

2. CNN Dimensions (Calculation):

- **Task:** Write a helper function to calculate the spatial dimensions of a feature map after a Convolutional layer.
- **Scenario:** Input Image 28×28, Filter 5×5, Stride $S = 1$, Padding $P = 0$. What is the output size?.

3. ICA vs. PCA (Blind Source Separation):

- **Task:** Use 'fastICA' to separate mixed signals. Compare conceptually with PCA.

1. Linear Discriminant Analysis (Manual Implementation):

- ***Task:*** *Implement a function to calculate the LDA discriminant score $\delta_k(x)$ manually (without using 'MASS::lda' prediction).*
- ***Theory:*** *LDA maximizes separation between classes by assuming Gaussian distributions with a shared covariance matrix Σ.*
- ***Formula:*** $\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\pi_k)$.

2. CNN Dimensions (Calculation):

- ***Task:*** *Write a helper function to calculate the spatial dimensions of a feature map after a Convolutional layer.*
- ***Scenario:*** *Input Image 28×28, Filter 5×5, Stride $S = 1$, Padding $P = 0$. What is the output size?.*

3. ICA vs. PCA (Blind Source Separation):

- ***Task:*** *Use 'fastICA' to separate mixed signals. Compare conceptually with PCA.*

```
# ---------------------------------------------------------
# 1. MANUAL LDA CLASSIFIER
# ---------------------------------------------------------
# Function to calculate LDA Score for class k
# x: New observation (vector)
# mu_k: Mean vector for class k
# Sigma_inv: Inverse of common Covariance Matrix (solve(Sigma))
# pi_k: Prior probability of class k
lda_score <- function(x, mu_k, Sigma_inv, pi_k) {
  term1 <- t(x) %*% Sigma_inv %*% mu_k
  term2 <- 0.5 * t(mu_k) %*% Sigma_inv %*% mu_k
  term3 <- log(pi_k)
  return(as.numeric(term1 - term2 + term3))
}

# Example Usage (Hypothetical Data):
# mu1 <- c(1,1); mu2 <- c(4,4); Sigma <- matrix(c(1,0,0,1),2)
# Sigma_inv <- solve(Sigma)
# x_new <- c(2,2)
# s1 <- lda_score(x_new, mu1, Sigma_inv, 0.5)
# s2 <- lda_score(x_new, mu2, Sigma_inv, 0.5)
# Prediction <- ifelse(s1 > s2, "Class 1", "Class 2")

# ----------------------------------------------------------
# 2. CNN OUTPUT DIMENSION CALCULATOR
# ----------------------------------------------------------
# W: Input Width/Height
# F: Filter Size
# P: Padding
# S: Stride
calc_cnn_dim <- function(W, F, P, S) {
  # Formula: floor((W - F + 2*P) / S) + 1
  return(floor((W - F + 2*P) / S) + 1)
}

# Example Test:
# Input 28x28, Filter 5x5, Pad 0, Stride 1
# out_dim <- calc_cnn_dim(28, 5, 0, 1)
# print(out_dim) # Result should be 24

# ----------------------------------------------------------
# 3. ICA SYNTAX (Independent Component Analysis)
# ----------------------------------------------------------
# library(fastICA)
# Requires data matrix X where rows are observations
# model <- fastICA(X, n.comp = 3, alg.typ = "parallel",
#                  fun = "logcosh", alpha = 1, method = "R")
# S_est <- model$S  # The independent sources (Separated signals)
# A_est <- model$A  # The mixing matrix
```

Analysis & Theory:

- **LDA vs Logistic:** LDA assumes data is Gaussian. If true, LDA is more efficient. Logistic Regression is more robust to non-Gaussian data.
- **CNNs:** Convolutions preserve spatial relationships. *Pooling* reduces dimensions. *Filters* extract features (edges/textures).
- **ICA vs PCA:** PCA decorrelates data (Second-order statistics, Gaussian assumption). ICA seeks *statistical independence* (Higher-order statistics, Non-Gaussian). Use ICA for ”Blind Source Separation” (e.g., Cocktail Party Problem).

Theory Gaps (From Slides)

- **ICA (Indep. Component Analysis):** Finds components that are statistically independent and non-Gaussian. Unlike PCA (which decorrelates), ICA recovers original sources (e.g., separating voices).

- **Probabilistic PCA:** PCA formulated as a Latent Variable model $x \sim \mathcal{N}(Wz + \mu, \sigma^2 I)$. Handles missing data better than standard PCA.

- **CNNs:** Specialized for grids (images). Convolution: Detects local patterns (edges). Pooling: Reduces size (translation invariance).

- **LDA vs Logistic:** LDA maximizes class separation (between-class var / within-class var). Assumes Gaussian classes with same covariance. Logistic is more robust to non-Gaussian data.

Handling Loss Matrices

1. Map Matrix Values to Costs
Locate the non-zero numbers in the assignment matrix:

		Prediction	
		”Male” (Pos)	”Female” (Neg)
2***True**	”Male”	0	C_{FN} (Top Right)
	”Female”	C_{FP} (Bottom Left)	0

Example from image: $C_{FN} = 1$ (Missed Male), $C_{FP} = 10$ (False Alarm).

2. Calculate Threshold Formula

$$Threshold(\tau) = \frac{C_{FP}}{C_{FP} + C_{FN}}$$

3. R Code Snippet

```
# 1. INPUT: Copy values directly from the exam matrix positions
# Top-Right number in the grid (Cost of predicting Neg when True Pos)
cost_FN <- 1

# Bottom-Left number in the grid (Cost of predicting Pos when True Neg)
cost_FP <- 10

# 2. CALC: Compute the optimal probability threshold
# If Cost_FP is high (10), threshold will be high (>0.9) to avoid false alarms.
threshold <- cost_FP / (cost_FP + cost_FN)

# 3. PREDICT: Apply threshold to model probabilities
# "If prob > threshold, predict Male"
preds <- ifelse(probs > threshold, "Male", "Female")
```

Log Reg probabilsitic equation

Probabilistic Equation

The model estimates the probability of a patient having diabetes (P(Diabetes = 1)) using the following logistic function. The coefficients ($\beta_0, \beta_1, \beta_2$) were obtained from the model's summary, which is ex: betas <- coef(modelglm), and then print it:

$$P(Diabetes = 1) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 \cdot Glucose + \beta_2 \cdot Age))}$$

Where the estimated parameters are:

- β_0 (Intercept): -5.897858
- β_1 (Glucose): 0.035582
- β_2 (Age): 0.024502

Exam 2022, hold out, decision tree

Problem Statement:
1. Divide data, use hold out principle, grow decision tree with default settings. Target income level. Exclude native country. Compute: Decision trees (many leaves), optimal tree according to deviance criterion. Plot dependence of training and validation errors (bias variance tradeoff). 2. Estimate predictions for different thresholds (account for too many of one category in features). Which is best threshold, look at F1? 3. Cross validation error (lasso) plot, optimal penalty, report equation of lasso model (include coef(cv$_lasso, s = "lambda.min")forcoefs)$.

```
data <- read.csv("adult.csv", stringsAsFactors = TRUE)        # Load data, treat strings as
    ↪ factors
colnames(data) <- c("age", "workclass", "finalweight",          # Rename columns for readability
                    "education", "education_num", "martial_status", "occupation",
                    "relationship", "capital_gain", "capital_loss",
                    "hours_per_week", "native_country", "income_level")
data <- subset(data, select=-native_country)                    # Remove 'native_country' column
n <- dim(data)[1]                                               # Count total rows
set.seed(12345)                                                 # Set seed for reproducibility
```

```
id <- sample(1:n, floor(n*0.6))                               # Sample 60% indices for training
id1 <- setdiff(1:n, id)                                       # Identify remaining indices
id2 <- sample(id1, floor(n*0.2))                              # Sample 20% for validation
id3 <- setdiff(id1, id2)                                      # Identify remaining 20% for
    ↪ testing
train <- data[id, ]                                           # Create training dataframe
valid <- data[id2, ]                                          # Create validation dataframe
test <- data[id3, ]                                           # Create test dataframe
library(tree)                                                 # Load tree package
treeDefault <- tree(income_level ~., data=train)              # Fit default classification tree
plot(treeDefault)                                             # Plot the tree structure
text(treeDefault, pretty=0)                                   # Add text labels to the plot
treeDefault                                                   # Print tree details
summary(treeDefault)                                          # Print tree summary statistics
trainScore <- rep(0,8)                                        # Initialize vector for train
    ↪ scores
validScore <- rep(0,8)                                        # Initialize vector for valid
    ↪ scores
for (i in 2:8) {                                              # Loop through tree sizes 2 to 8
  treePruned <- prune.tree(treeDefault, best=i)               # Prune tree to 'i' terminal nodes
  pred <- predict(treePruned, newdata=valid, type='tree')     # Predict on validation set
  trainScore[i] <- deviance(treePruned)                       # Store training deviance
  validScore[i] <- deviance(pred)                             # Store validation deviance
}
plot(2:8, trainScore[2:8], type='b', col='red', ylim = c(0, 20000)) # Plot training deviance
points(2:8, validScore[2:8], type='b', col='blue', ylim = c(0, 20000)) # Add validation deviance
legend("topright",                                            # Add legend to the plot
       legend=c("Training deviance", "Validation deviance"),
       col=c("red", "blue"), lty=1, pch=20)
treeOpt <- prune.tree(treeDefault, best=8)                    # Create optimal tree (size 8)
plot(treeOpt)                                                 # Plot optimal tree
text(treeOpt, pretty=0)                                       # Label optimal tree
#======== 1.2 ============                                     # Section 1.2 marker
predTest <- predict(treeOpt, newdata=test, type='class')      # Predict classes on test set
confMat <- table(Actual = test$income_level, Predicted = predTest) # Create confusion matrix
confMat                                                       # Print confusion matrix
accuracy <- sum(diag(confMat)/sum(confMat))                   # Calculate accuracy
accuracy                                                      # Print accuracy
TP <- confMat[2, 2]                                           # Extract True Positives
TP                                                            # Print TP
FP <- confMat[1, 2]                                           # Extract False Positives
FN <- confMat[2, 1]                                           # Extract False Negatives
precision <- TP / (TP+FP)                                     # Calculate Precision
recall <- TP / (TP+FN)                                        # Calculate Recall
F1 <- 2 * precision * recall / (precision+recall)             # Calculate F1 Score
F1                                                            # Print F1 Score
tree_probs <- predict(treeOpt, test, type="vector")[, 2]      # Get probabilities for positive
    ↪ class
thresholds <- seq(0.1, 0.9, by=0.1)                           # Define sequence of thresholds
getTRPFPR <- function(probs, true_y, thresh){                 # Define function for TPR/FPR
    ↪ stats
  pred <- ifelse(probs > thresh, 2, 1)                        # Classify based on threshold
  pred <- factor(pred, levels=c(1, 2))                        # Convert to factor
  cm <- table(Predicted=pred, Actual=true_y)                  # Make local confusion matrix
  TP <- cm[2,2]; FP <- cm[2, 1]                               # Extract TP and FP
  FN <- cm[1, 2]; TN <- cm[1, 1]                              # Extract FN and TN
  TPR <- TP / (TP+FN); FPR <- FP / (FP+TN)                    # Calculate rates
  F1 <- 2 * TPR * FPR / (TPR+FPR)                             # Calculate F1 for this thresh
  c(TPR = TPR, FPR = FPR, F1 = F1)                            # Return stats
}
tree_roc <- t(sapply(thresholds, function(t){                 # Apply function over thresholds
  getTRPFPR(tree_probs,test$income_level,t)
}))
tree_roc                                                        # Print ROC/F1 table
#0.3 is the correct threshold                                   # Note on optimal threshold
#========= 1.3 ============                                      # Section 1.3 marker
library(glmnet)                                                 # Load glmnet package
vars <- c("age", "capital_gain", "capital_loss")                # Define predictors list
x_train <- as.matrix(train[, vars])                             # Convert train predictors to
    ↪ matrix
y_train <- train$hours_per_week                                 # Extract train target
set.seed(12345)                                                 # Set seed for CV
cv_lasso <- cv.glmnet(x_train, y_train, alpha = 1, family = "gaussian") # Run Cross-Validation
    ↪ Lasso
plot(cv_lasso)                                                  # Plot CV error vs Lambda
best_lambda <- cv_lasso$lambda.min                              # Extract best lambda
print(best_lambda)                                              # Print best lambda
print(log(best_lambda))                                         # Print log of best lambda
coef_optimal <- coef(cv_lasso, s = "lambda.min")                # Extract coefficients at best
    ↪ lambda
num_vars <- sum(coef_optimal != 0) -1                           # Count non-zero coefs (minus
    ↪ intercept)
print(num_vars)                                                 # Print count of variables
```

Exam 2023-01-13, basis function expansion, logreg, decision trees

Problem Statement:
1. Divide data, fit log reg on features, fit log reg on basis function expansion, training and misclassification errors, probabilistic model.
2. Cross validation to fit decision tree. Report optimal number of leaves and training and test misclassification errors for the optimal tree. Prediction quality? Loop that grows decision tree with mindev 0.001, 0.002,..0.01 (without cv). Estimate training and test misclassification errors for these. Plot the dependence of these errors on mindev.
3. From step 2: if we use missclassification error as impurity measure, which leaves must be pruned first, 14 and 15, or 4 and 5? Report math calcs.

```
data <- read.csv("Rice.csv")                                    # Load the Rice dataset
data$Class <- as.factor(data$Class)                             # Convert Class to factor before
    ↪ splitting
set.seed(12345)                                                 # Set seed for reproducibility
n <- nrow(data)                                                 # Get total number of rows
train_id <- sample(1:n, floor(0.7 * n))                         # Sample 70% indices for training
train_data <- data[train_id, ]                                  # Create training set
test_data <- data[-train_id, ]                                  # Create test set
model_glm <- glm(Class ~ ., family = "binomial", data = train_data) # Fit logistic regression on
    ↪ training data
probs_train <- predict(model_glm, newdata = train_data, type = "response") # Get training
    ↪ probabilities
pos_class <- levels(train_data$Class)[2]                        # Identify "Positive" class name (
    ↪ usually 2nd level)
neg_class <- levels(train_data$Class)[1]                        # Identify "Negative" class name
preds_train <- ifelse(probs_train > 0.5, pos_class, neg_class) # Predict class labels using 0.5
    ↪ threshold
error_train <- mean(preds_train != train_data$Class)            # Calculate training
    ↪ misclassification error
print(paste("Training Misclassification Error:", error_train)) # Print training error
probs_test <- predict(model_glm, newdata = test_data, type = "response") # Get test probabilities
preds_test <- ifelse(probs_test > 0.5, pos_class, neg_class)   # Predict test class labels
error_test <- mean(preds_test != test_data$Class)              # Calculate test misclassification
    ↪  error
print(paste("Test Misclassification Error:", error_test))      # Print test error
betas <- coef(model_glm)                                        # Extract model coefficients
```

```
print(betas)                                               # Print coefficients
#-------- basis function expansion------
train_data$z1 <- train_data$Area^2                         # Expand feature: Area^2
train_data$z2 <- train_data$Perimeter^2                    # Expand feature: Perimeter^2
train_data$z3 <- train_data$Major_Axis_Length^2            # Expand feature: Major_Axis^2
train_data$z4 <- train_data$Minor_Axis_Length^2            # Expand feature: Minor_Axis^2
train_data$z5 <- train_data$Eccentricity^2                 # Expand feature: Eccentricity^2
train_data$z6 <- train_data$Convex_Area^2                  # Expand feature: Convex_Area^2
train_data$z7 <- train_data$Extent^2                       # Expand feature: Extent^2
test_data$z1 <- test_data$Area^2                           # Apply same expansion to test
    ↪ data
test_data$z2 <- test_data$Perimeter^2                      # Apply same expansion to test
    ↪ data
test_data$z3 <- test_data$Major_Axis_Length^2              # Apply same expansion to test
    ↪ data
test_data$z4 <- test_data$Minor_Axis_Length^2              # Apply same expansion to test
    ↪ data
test_data$z5 <- test_data$Eccentricity^2                   # Apply same expansion to test
    ↪ data
test_data$z6 <- test_data$Convex_Area^2                    # Apply same expansion to test
    ↪ data
test_data$z7 <- test_data$Extent^2                         # Apply same expansion to test
    ↪ data
model_expanded <- glm(Class ~ ., family = "binomial", data = train_data) # Fit model on expanded
    ↪ data
probs_exp <- predict(model_expanded, newdata = train_data, type = "response") # Get expanded
    ↪ training probs
preds_train_exp <- ifelse(probs_exp > 0.5, pos_class, neg_class) # Predict expanded training
    ↪ labels
error_train_exp <- mean(preds_train_exp != train_data$Class)   # Calculate expanded training
    ↪ error
print(paste("Training Misclassification Error:", error_train_exp)) # Print expanded training
    ↪ error
probs_test_exp <- predict(model_expanded, newdata = test_data, type = "response") # Get expanded
    ↪ test probs
preds_test_exp <- ifelse(probs_test_exp > 0.5, pos_class, neg_class) # Predict expanded test
    ↪ labels
error_test_exp <- mean(preds_test_exp != test_data$Class)      # Calculate expanded test error
print(paste("Test Misclassification Error:", error_test_exp))  # Print expanded test error
# -------------- 1.2 decision tree
library(tree)                                              # Load tree library
tree_model <- tree(Class ~ ., data = train_data)           # Fit decision tree
cv_tree <- prune.tree(tree_model, method = "deviance")     # Run Cross-Validation using
    ↪ deviance
test_deviance <- cv_tree$dev                               # Extract deviance values
opt_leaves <- cv_tree$size[which.min(test_deviance)]       # Find size with minimum deviance
cat("Optimal number of leaves:", opt_leaves, "\n")         # Print optimal leaf count
summary(cv_tree)                                           # Summarize CV results
plot(tree_model)                                           # Plot the full tree
optimal_tree <- prune.tree(tree_model, best = opt_leaves)  # Prune tree to optimal size
misClassError <- function(tree, newdata){                  # Define error calculation
    ↪ function
  pred <- predict(tree, newdata=newdata, type='class')     # Predict class
  mean(pred != newdata$Class)                              # Return misclassification rate
}
mis_train <- misClassError(optimal_tree, train_data)       # Calculate optimal tree train
    ↪ error
mis_test <- misClassError(optimal_tree, test_data)         # Calculate optimal tree test
    ↪ error
cat("Misscalssification error on training data:", mis_train, "\n") # Print train error
cat("Misscalssification error on test data:", mis_test, "\n")  # Print test error
# ----------- grow tree with mindev
```

```
mindev_values <- seq(from = 0.001, to = 0.01, by = 0.001)      # Define sequence of mindev values
train_errors <- numeric(length(mindev_values))                 # Init train error vector
test_errors  <- numeric(length(mindev_values))                 # Init test error vector
tree_sizes   <- numeric(length(mindev_values))                 # Init tree size vector
# --- looop ----
for (i in 1:length(mindev_values)) {                           # Loop through mindev values (
    ↪ First Pass)
  current_mindev <- mindev_values[i]                           # Get current mindev
  fit <- tree(Class ~ ., data = train_data,                    # Fit tree with control params
              control = tree.control(nobs = nrow(train_data), mindev = current_mindev))
  train_errors[i] <- misClassError(fit, train_data)            # Store train error
  test_errors[i]  <- misClassError(fit, test_data)             # Store test error
  tree_sizes[i]   <- summary(fit)$size                         # Store tree size
}
mindev_values <- seq(from = 0.001, to = 0.01, by = 0.001)      # Re-define mindev values (
    ↪ Redundant)
train_errors <- numeric(length(mindev_values))                 # Re-init train error vector
test_errors  <- numeric(length(mindev_values))                 # Re-init test error vector
tree_sizes   <- numeric(length(mindev_values))                 # Re-init tree size vector
for (i in 1:length(mindev_values)) {                           # Loop through mindev values (
    ↪ Second Pass)
  current_mindev <- mindev_values[i]                           # Get current mindev
  fit <- tree(Class ~ ., data = train_data,                    # Fit tree with control params
              control = tree.control(nobs = nrow(train_data), mindev = current_mindev))
  train_errors[i] <- misClassError(fit, train_data)            # Store train error
  test_errors[i]  <- misClassError(fit, test_data)             # Store test error
  tree_sizes[i]   <- summary(fit)$size                         # Store tree size
}
plot(mindev_values, train_errors, type = "b", col = "blue",    # Plot Training Error (Blue)
     ylim = range(c(train_errors, test_errors)),               # Set Y limits to fit both lines
     xlab = "mindev parameter", ylab = "Misclassification Error", # Labels
     main = "Effect of mindev on Error Rates")                 # Title
lines(mindev_values, test_errors, type = "b", col = "red")     # Add Test Error line (Red)
legend("topright", legend = c("Training Error", "Test Error"), # Add Legend
       col = c("blue", "red"), lty = 1, pch = 1)
#----- 1.3 impurtiy ---
print(optimal_tree)                                            # Print details of optimal tree
ImpurityDiff0=0.057796*1488-(0.011649*1116  +0.196237*372)     # Manual calculation (Node 0?)
ImpurityDiff0                                                  # Print result
ImpurityDiff1=0.026504*981  -(0.061125*409+0.001748*572 )      # Manual calculation (Node 1?)
ImpurityDiff1                                                  # Print result
```

Part 1: Probabilistic model is: P(Class = Osmanick) = 1/(1 + exp(z))
where z = 7.910612 - 0.006971 Area - 0.107500 Perimeter + 0.112395 MajorAxisLength - 0.553997 MinorAxisLength + 3.747786 Eccentricity + 0.011754 ConvexArea - 0.315431 Extent
Part 2: Training error increasing with increasing mindev, and test error decrases. In theory, increasing mindev makes it harder to split nodes which leads to smaller trees (less complex ones). Inceeasing mindev -> decreasing tree complexity which means training error should go up. If model was overfitted, test error will go down.
Part 3: 4 and 5 need to be pruned first since it provides the smallest impurity decrease

Formulas and Cost Functions from TDDE01 Exams

1. Probabilistic and Prediction Models

Logistic Regression Probabilistic Model

Used for binary classification (e.g., Rice Data, Cell Types). P(Y=1 | **x**) $= \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\ldots+\beta_p x_p)}}$

Linear / LASSO Prediction Equation

Used for regression (e.g., Bikes, WDBC, Adult). $\hat{y} = \beta_0 + \sum_{j\in\mathcal{S}} \beta_j x_j$ *Note: In LASSO, only selected features with non-zero coefficients are included.*

PCA Reconstruction Approximation

Approximating a variable (e.g., `Cond`) from the first two PCs. $x_j \approx \mu_j + \sigma_j(v_{j,1}z_1 + v_{j,2}z_2)$ Where μ, σ are scaling parameters and v are loadings.

First Principal Component

Equation of PC1 in terms of scaled features. z$_1 = w_1 x_{1,scaled} + \ldots + w_p x_{p,scaled}$

2. Cost and Objective Functions

Binary Logistic Regression Cost (Negative Log-Likelihood)

Objective function to minimize. J$(\theta) = -\sum_{i=1}^{N} [y_i \log(h_\theta(x_i)) + (1-y_i)\log(1-h_\theta(x_i))]$

Hinge Loss Cost Function (SVM)

For model $\hat{y} = sign(\mathbf{w}^T\mathbf{x})$ with $y \in \{-1, 1\}$. J(w) = $\sum_{i=1}^{N} \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i)\right) + \lambda||\mathbf{w}||^2$

Mean Absolute Error (MAE) Cost

Robust regression cost function. J$(\theta) = \sum_{i=1}^{N} |y_i - \hat{y}_i|$

Heteroscedastic Negative Log-Likelihood

For model $y \sim \mathcal{N}(\mu(\mathbf{x}, \mathbf{w}), \sigma^2(\mathbf{x}))$ (e.g., Australian Crabs). -ln $L(\mathbf{w}) \propto \sum_{i=1}^{N} \left(\ln(\sigma^2(x_i)) + \frac{(y_i - \mu(x_i, \mathbf{w}))^2}{\sigma^2(x_i)}\right)$

3. Density Estimation and Kernel Methods

Kernel Density Estimator (Gaussian Kernel)

Estimating density $p(x_*)$. p(x$_*) = \frac{1}{Nh}\sum_{i=1}^{N} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x_* - x_i}{h}\right)^2}$

Bayes' Theorem for Classification

Posterior probability using class-conditional densities. P(C=1 $| \; x) = \frac{p(x|C=1)P(C=1)}{p(x|C=1)P(C=1) + p(x|C=2)P(C=2)}$

4. Algorithm Updates

Perceptron Update Rule

For misclassified point (x_n, t_n). w$^{(new)} = \mathbf{w}^{(old)} + \alpha(t_n - y(\mathbf{w}^{(old)}, x_n))x_n$

Neural Network Backpropagation (Output Layer)

Gradient for output weights $W^{(2)}$ (MSE Loss). $\partial J \frac{}{\partial W^{(2)} = \delta^{(2)}(q^{(1)})^T, \quad where \delta^{(2)} = -2(y - z^{(2)})}$