ml help

Varun Gurupurandar

2025-06-26

PCA GENERALIZED CODE

```
library(ggplot2) # Load necessary libraries
library(caret)
# Function to scale data (except the target variable)
scale_data <- function(data, target_col) {
  scaled_data <- data
  scaled_data[, -which(names(data) == target_col)] <- scale(data[, -which(names(data) == target_col)])
  return(scaled_data)
}# Function to perform PCA and return results
perform_pca <- function(data, target_col, scale_data = TRUE) {
  if (scale_data) {
    data <- scale_data(data, target_col)
  }
   pca_result <- prcomp(data[, -which(names(data) == target_col)], center = TRUE, scale. = scale_data)
  return(pca_result)
}# Function to calculate the proportion of variance explained by PCA components
pca_variance <- function(pca_result) {
  variance <- pca_result$sdev^2 / sum(pca_result$sdev^2)
  return(variance)
}# Function to plot PCA results
plot_pca <- function(pca_result, data, target_col) {
  pca_scores <- as.data.frame(pca_result$x)
  pca_scores[[target_col]] <- data[[target_col]]
  ggplot(pca_scores, aes(x = PC1, y = PC2, color = as.factor(get(target_col)))) +
    geom_point() +
    labs(title = "PCA Scatter Plot", x = "PC1", y = "PC2", color = target_col)
}# Function to compute the number of components needed to explain >95% variance
pca_components_95 <- function(pca_result) {
  variance <- pca_variance(pca_result)
  cumulative_variance <- cumsum(variance)
  num_components <- which(cumulative_variance > 0.95)[1]
  return(num_components)
}# Function to reconstruct data after PCA compression
reconstruct_data <- function(pca_result, num_components, original_data) {
  compressed_data <- pca_result$x[, 1:num_components]
  reconstruction <- compressed_data %*% t(pca_result$rotation[, 1:num_components])
  return(reconstruction)
}# Function to perform KNN regression using base R
knn_regression <- function(train_data, test_data, target_col, k) {
  train_features <- train_data[, -which(names(train_data) == target_col)]
  test_features <- test_data[, -which(names(test_data) == target_col)]
```

```
  train_target <- train_data[[target_col]]
  predictions <- sapply(1:nrow(test_features), function(i) {
    distances <- sqrt(rowSums((train_features - test_features[i, ])^2))
    nearest_neighbors <- order(distances)[1:k]
    mean(train_target[nearest_neighbors])
  })
  return(predictions)
}# Problem 1: Female cancer patients # Example usage for each problem statement
data1 <- read.csv("female_cancer_data.csv")
pca_result1 <- perform_pca(data1, "Death")
variance1 <- pca_variance(pca_result1)
cat("Variance explained by first two components:", sum(variance1[1:2]), "\n")
plot_pca(pca_result1, data1, "Death")
# Problem 2: Handwritten digits
data2 <- read.csv("handwritten_digits.csv")
pca_result2 <- perform_pca(data2, "Digit", scale_data = FALSE)
num_components2 <- pca_components_95(pca_result2)
cat("Number of components for >95% variance:", num_components2, "\n")
reconstructed_data2 <- reconstruct_data(pca_result2, num_components2, data2)
# Problem 3: Glass types
data3 <- read.csv("glass_data.csv")
pca_result3 <- perform_pca(data3, "Class")
variance3 <- pca_variance(pca_result3)
cat("Variance explained by first two components:", sum(variance3[1:2]), "\n")
# Problem 4: Spectrographic analysis
data4 <- read.csv("spectrographic_data.csv")
pca_result4 <- perform_pca(data4, "Fat", scale_data = FALSE)
pca_scores4 <- as.data.frame(pca_result4$x)
pca_scores4$Fat <- data4$Fat
# Split data into training and test sets
set.seed(123)
train_index <- createDataPartition(pca_scores4$Fat, p = 0.8, list = FALSE)
train_data <- pca_scores4[train_index, ]
test_data <- pca_scores4[-train_index, ]
# KNN regression
knn_predictions <- knn_regression(train_data, test_data, "Fat", k = 5)
train_mse <- mean((train_data$Fat - knn_predictions)^2)
test_mse <- mean((test_data$Fat - knn_predictions)^2)
cat("Training MSE:", train_mse, "\n")
cat("Test MSE:", test_mse, "\n")
```

DECISION TREE WITH ALL THE POSSIBLE APPROACH

```
# Load necessary libraries
library(tree)         # For decision trees
library(caret)        # For data splitting and model evaluation
library(ggplot2)      # For plotting
library(dplyr)        # For data manipulation
set.seed(123) # Set seed for reproducibility
# Function to split data into training and test sets
split_data <- function(data, target, train_ratio) {
  train_index <- createDataPartition(data[[target]], p = train_ratio, list = FALSE)
  train_data <- data[train_index, ]
```

```
  test_data <- data[-train_index, ]
  return(list(train = train_data, test = test_data))
}
# Function to fit decision trees with varying complexity
fit_decision_tree <- function(data, target, max_leaves) {
  train_errors <- c()
  test_errors <- c()
  for (leaves in 1:max_leaves) {
    tree_model <- tree(as.formula(paste(target, "~ .")), data = data$train,
                       control = tree.control(nobs = nrow(data$train), minsize = 2, mindev = 0.01))
    # Calculate cross-entropy for training and test data
    train_pred <- predict(tree_model, data$train, type = "vector")
    train_errors[leaves] <- -mean(log(train_pred[cbind(1:nrow(data$train), as.numeric(data$train[[targe
    test_pred <- predict(tree_model, data$test, type = "vector")
    test_errors[leaves] <- -mean(log(test_pred[cbind(1:nrow(data$test), as.numeric(data$test[[target]])
  }
  return(list(train_errors = train_errors, test_errors = test_errors))
}# Function for basis expansion
basis_expansion <- function(data) {
  p <- ncol(data) - 1  # Exclude the target column
  expanded_data <- data
  for (i in 1:p) {
    expanded_data[[paste0("x", i, "_squared")]] <- data[[i]]^2
  }
  return(expanded_data)
}
# Function to fit logistic regression models
fit_logistic_regression <- function(train_data, test_data, target, expanded = FALSE) {
  if (expanded) {
    train_data <- basis_expansion(train_data)
    test_data <- basis_expansion(test_data)
  }
  model <- glm(as.formula(paste(target, "~ .")), data = train_data, family = binomial)
  train_pred <- predict(model, train_data, type = "response")
  test_pred <- predict(model, test_data, type = "response")
  train_error <- mean((train_pred > 0.5) != train_data[[target]])
  test_error <- mean((test_pred > 0.5) != test_data[[target]])
  return(list(model = model, train_error = train_error, test_error = test_error))
}# Function to perform cross-validation for decision trees
cross_validate_tree <- function(data, target) {
  tree_model <- tree(as.formula(paste(target, "~ .")), data = data$train,
                     control = tree.control(nobs = nrow(data$train), minsize = 2, mindev = 0.01))
  cv_results <- cv.tree(tree_model, FUN = prune.misclass)
  return(cv_results)
}# Function to prune decision tree
prune_decision_tree <- function(tree_model, optimal_size) {
  pruned_tree <- prune.tree(tree_model, best = optimal_size)
  return(pruned_tree)
}# Function to evaluate decision tree
evaluate_tree <- function(tree_model, train_data, test_data, target) {
  train_pred <- predict(tree_model, train_data, type = "class")
  test_pred <- predict(tree_model, test_data, type = "class")
  train_error <- mean(train_pred != train_data[[target]])
```

```
  test_error <- mean(test_pred != test_data[[target]])
  return(list(train_error = train_error, test_error = test_error))
}
# Main Execution
# Step 1: Load your dataset (replace `your_data` with your dataset)
# Example: crabs_data or rice_data
data <- your_data
target <- "Target_Column_Name"  # Replace with your target column name
# Step 2: Split data into training and test sets
split_ratio <- 0.7  # Adjust as needed
data_split <- split_data(data, target, split_ratio)
# Step 3: Fit decision trees and plot cross-entropy vs. number of leaves
max_leaves <- 20
tree_results <- fit_decision_tree(data_split, target, max_leaves)
# Plot results
ggplot(data.frame(Leaves = 1:max_leaves, Train = tree_results$train_errors, Test = tree_results$test_er
  geom_line(aes(y = Train, color = "Train")) +
  geom_line(aes(y = Test, color = "Test")) +
  labs(title = "Cross-Entropy vs. Number of Leaves", y = "Cross-Entropy", color = "Dataset") +
  theme_minimal()
# Step 4: Fit logistic regression models (with and without basis expansion)
logistic_model1 <- fit_logistic_regression(data_split$train, data_split$test, target, expanded = FALSE)
logistic_model2 <- fit_logistic_regression(data_split$train, data_split$test, target, expanded = TRUE)
cat("Logistic Regression (Original Features): Train Error =", logistic_model1$train_error, "Test Error =
cat("Logistic Regression (Expanded Features): Train Error =", logistic_model2$train_error, "Test Error =
# Step 5: Cross-validate decision tree
cv_results <- cross_validate_tree(data_split, target)
print(cv_results)
# Step 6: Prune and evaluate the optimal tree
optimal_size <- cv_results$size[which.min(cv_results$dev)]
optimal_tree <- prune_decision_tree(tree(as.formula(paste(target, "~ .")), data = data_split$train), op
tree_evaluation <- evaluate_tree(optimal_tree, data_split$train, data_split$test, target)
cat("Optimal Tree: Train Error =", tree_evaluation$train_error, "Test Error =", tree_evaluation$test_er
# Step 7: Pruning based on misclassification error
# (Replace leaf numbers with actual leaf IDs from your tree)
leaves_to_prune <- c(4, 5, 14, 15)  # Example leaf IDs
cat("Leaves to prune first:", leaves_to_prune, "\n")
```

LOGISTIC REGRESSION GENERALIZED CODE

```
library(caret) # Load necessary libraries
mydata <- read.csv("your_file.csv")# Load data (replace 'your_file.csv' actual filename)
# Function to perform logistic regression and evaluation
logistic_regression_analysis <- function(data, target, features, train_ratio=0.7, val_ratio=0.3) {
  set.seed(123)  # For reproducibility
  # Split data into train, validation, and test
  total <- nrow(data)
  train_idx <- sample(1:total, size = floor(train_ratio * total))
  remain_idx <- setdiff(1:total, train_idx)
  val_idx <- sample(remain_idx, size = floor(val_ratio * length(remain_idx)))
  test_idx <- setdiff(remain_idx, val_idx)
  train_data <- data[train_idx, ]
  val_data <- data[val_idx, ]
```

```
  test_data <- data[test_idx, ]
   # Fit logistic regression model
  formula <- as.formula(paste(target, "~", paste(features, collapse = "+")))
  model <- glm(formula, data = train_data, family = binomial
  # Summary of the model
  print(summary(model))
  # Predict and evaluate on validation data
  val_pred <- predict(model, val_data, type = "response")
  val_class <- ifelse(val_pred > 0.5, 1, 0)
  confusion <- table(val_data[[target]], val_class)
  print("Confusion Matrix (Validation):")
  print(confusion)
  misclassification_error <- mean(val_class != val_data[[target]])
  print(paste("Validation Misclassification Error:", misclassification_error))
  # Compute F1 Score
  precision <- sum(val_class == 1 & val_data[[target]] == 1) / sum(val_class == 1)
  recall <- sum(val_class == 1 & val_data[[target]] == 1) / sum(val_data[[target]] == 1)
  f1_score <- 2 * (precision * recall) / (precision + recall)
  print(paste("F1 Score:", f1_score))
  return(model)
}# Example Usage
# Assume 'mydata' is the dataset, 'Sex' is the target, and 'CW', 'BD' are features
logistic_model <- logistic_regression_analysis(mydata, "Sex", c("CW", "BD"))
# Function to compute minus log-likelihood
minus_log_likelihood <- function(model, data) {
  predictions <- predict(model, data, type = "response")
  y <- data[[as.character(formula(model)[[2]])]]
  return(-sum(y * log(predictions) + (1 - y) * log(1 - predictions)))
}# Function to perform logistic regression with principal component
logistic_with_pca <- function(data, target) {
  pca <- prcomp(data[, -which(names(data) == target)], scale. = TRUE)
  data$PC1 <- pca$x[, 1]
  model <- glm(as.formula(paste(target, "~ PC1")), data = data, family = binomial)
  print(summary(model))
  # Plot probabilities
  pc1_vals <- seq(min(data$PC1), max(data$PC1), length.out = 100)
  pred <- predict(model, newdata = data.frame(PC1 = pc1_vals), type = "response")
  plot(pc1_vals, pred, type = "l", col = "blue", xlab = "PC1", ylab = "Probability (Y=1)")
}# Example usage with digit data
# logistic_with_pca(digit_data, "Y")
```

KNN GENERALIZED CODE

```
library(caret)
library(kknn)
library(ggplot2)# Replace 'path/to/your/dataset.csv' with the actual file path# Load dataset
data <- read.csv("path/to/your/dataset.csv")
# Remove 'Death' variableOR any variable not needed
#data <- subset(data, select = -c(Death))
# General function for KNN modeling
general_knn <- function(data, target, k_values = 1:30, test_size = 0.5) {
  set.seed(123)  # For reproducibility
  # Split data into train and test sets
```

```
  train_index <- createDataPartition(data[[target]], p = 1 - test_size, list = FALSE)
  train_data <- data[train_index, ]
  test_data <- data[-train_index, ]
  mse_results <- sapply(k_values, function(k) {
    knn_model <- kknn(as.formula(paste(target, "~ .")), train_data, train_data, k = k, kernel = "rectang
    knn_model_test <- kknn(as.formula(paste(target, "~ .")), train_data, test_data, k = k, kernel = "re
    c(mean((fitted(knn_model) - train_data[[target]])^2), mean((fitted(knn_model_test) - test_data[[targ
  })
  mse_df <- data.frame(k = k_values, Train_MSE = mse_results[1, ], Test_MSE = mse_results[2, ])
  # Plot training and test errors
  ggplot(mse_df, aes(x = k)) +
    geom_line(aes(y = Train_MSE, color = "Training Error")) +
    geom_line(aes(y = Test_MSE, color = "Test Error")) +
    labs(title = "KNN Performance", x = "K", y = "MSE") +
    theme_minimal() +
    scale_color_manual(values = c("Training Error" = "blue", "Test Error" = "red"))
  return(mse_df[which.min(mse_df$Test_MSE), ])
}# Example usage
# result <- general_knn(data, target = "Fat")
# print(result)
```

LASSO AND RIDGE REGRESSION GENERALIZED CODE

```
library(glmnet) # Load your dataset
# Example: your_data <- read.csv("path/to/your_data.csv")
fit_model <- function(data, target, model_type = "lasso", split_ratio = 0.5) {
  set.seed(123)    # Split data
  trainIndex <- sample(1:nrow(data), size = split_ratio * nrow(data))
  train <- data[trainIndex, ]
  test <- data[-trainIndex, ]  # Prepare data
  x_train <- as.matrix(train[, !names(train) %in% target])
  y_train <- train[[target]]
  x_test <- as.matrix(test[, !names(test) %in% target])
  y_test <- test[[target]]
  # Set alpha (1 for LASSO, 0 for Ridge)
  alpha_val <- ifelse(model_type == "lasso", 1, 0)
  # Fit model with cross-validation
  cv_model <- cv.glmnet(x_train, y_train, alpha = alpha_val)
  best_lambda <- cv_model$lambda.min # CAN BE ALTERED WITH THIS lambda.1se
  final_model <- glmnet(x_train, y_train, alpha = alpha_val, lambda = best_lambda)  # Predictions
  y_pred <- predict(final_model, x_test)  # Output results
  list(lambda = best_lambda, coefficients = coef(final_model), predictions = y_pred)
}# Example Usage:
# your_data <- read.csv("path/to/your_data.csv")
# fit_model(your_data, target = "YourTargetColumn", model_type = "lasso")
```

GENERALIZED PCA

```
library(stats)       # For PCA
library(ggplot2)     # For visualization
# Step 1: Load your dataset
# Replace this with code to load your dataset (e.g., from a CSV file)
# Example: my_data <- read.csv("path/to/your/dataset.csv")
```

```r
# For demonstration, we'll use the built-in iris dataset
data(iris)
my_data <- iris
# Step 2: Inspect the dataset
# Check the structure and first few rows of the dataset
str(my_data)
head(my_data)
# Step 3: Prepare the data
# Select only numeric columns (PCA works only on numeric data)
numeric_data <- my_data[, sapply(my_data, is.numeric)]
# Handle missing values (if any)
# Option 1: Remove rows with missing values
numeric_data <- na.omit(numeric_data)
# Option 2: Impute missing values with the mean
# numeric_data[is.na(numeric_data)] <- colMeans(numeric_data, na.rm = TRUE)
# Step 4: Standardize the data
# Standardize the data so that each feature has a mean of 0 and a standard deviation of 1
scaled_data <- scale(numeric_data)
# Step 5: Perform PCA
# Fit PCA on the standardized data
pca_result <- prcomp(scaled_data, center = TRUE, scale. = TRUE)
# Step 6: Examine the PCA results
# Print the PCA summary
summary(pca_result)
# View the loadings (rotation matrix)
print(pca_result$rotation)
# View the principal components (transformed data)
head(pca_result$x)
# Step 7: Explained Variance
# Calculate the proportion of variance explained by each principal component
explained_variance <- pca_result$sdev^2 / sum(pca_result$sdev^2)
print(explained_variance)
# Step 8: Scree Plot
# Plot the explained variance to decide how many principal components to retain
plot(explained_variance, type = "b", xlab = "Principal Component", ylab = "Proportion of Variance Expla
# Step 9: Visualize the PCA results
# Create a data frame for visualization
pca_df <- as.data.frame(pca_result$x)
# If your dataset has a grouping variable (e.g., a target column), add it to the data frame
if ("Group" %in% colnames(my_data)) {
  pca_df$Group <- my_data$Group
}# Plot the first two principal components
if ("Group" %in% colnames(pca_df)) {
  ggplot(pca_df, aes(x = PC1, y = PC2, color = Group)) +
    geom_point(size = 3) +
    labs(title = "PCA of Dataset", x = "PC1", y = "PC2") +
    theme_minimal()
} else {
  ggplot(pca_df, aes(x = PC1, y = PC2)) +
    geom_point(size = 3) +
    labs(title = "PCA of Dataset", x = "PC1", y = "PC2") +
    theme_minimal()
}# Step 10: Interpretation of PCs
```

```r
# Examine the loadings to understand how original features contribute to each principal component
print(pca_result$rotation)
# Step 11: Determine the optimal number of components
# Calculate cumulative explained variance
cumulative_variance <- cumsum(explained_variance)
print(cumulative_variance)
```

LOGISTIC REGRESSION MORE GENERALIZED

```r
# Load necessary libraries
library(caret)       # For data splitting and model evaluation
library(pROC)        # For ROC curve and AUC
library(ggplot2)     # For visualization
# Step 1: Load your dataset
# Replace this with code to load your dataset (e.g., from a CSV file)
# Example: my_data <- read.csv("path/to/your/dataset.csv")
# For demonstration, we'll use the built-in mtcars dataset
data(mtcars)
my_data <- mtcars
# Step 2: Inspect the dataset
# Check the structure and first few rows of the dataset
str(my_data)
head(my_data)
# Step 3: Prepare the data
# Convert the target variable to a factor (logistic regression requires categorical target)
my_data$am <- as.factor(my_data$am)  # Assuming 'am' is the target variable
# Split the dataset into training and testing sets
set.seed(123)  # For reproducibility
train_index <- createDataPartition(my_data$am, p = 0.8, list = FALSE)
train_data <- my_data[train_index, ]
test_data <- my_data[-train_index, ]
# Step 4: Train the logistic regression model
# Use the glm() function with family = binomial for logistic regression
logistic_model <- glm(am ~ ., data = train_data, family = binomial)
# Step 5: Examine the model summary
# View the coefficients and statistical significance of predictors
summary(logistic_model)
# Step 6: Make predictions on the test set
# Predict probabilities
test_data$pred_prob <- predict(logistic_model, test_data, type = "response")
# Convert probabilities to class predictions (default threshold = 0.5)
test_data$pred_class <- ifelse(test_data$pred_prob > 0.5, 1, 0)
# Step 7: Evaluate the model
# Confusion matrix
confusion_matrix <- table(Actual = test_data$am, Predicted = test_data$pred_class)
print(confusion_matrix)
# Calculate accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", accuracy))
# Calculate precision, recall, and F1-score
precision <- confusion_matrix[2, 2] / sum(confusion_matrix[, 2])
recall <- confusion_matrix[2, 2] / sum(confusion_matrix[2, ])
f1_score <- 2 * (precision * recall) / (precision + recall)
```

```r
print(paste("Precision:", precision))
print(paste("Recall:", recall))
print(paste("F1-Score:", f1_score))
# ROC curve and AUC
roc_curve <- roc(test_data$am, test_data$pred_prob)
plot(roc_curve, main = "ROC Curve", col = "blue")
auc_value <- auc(roc_curve)
print(paste("AUC:", auc_value))
# Step 8: Visualize the results
# Plot predicted probabilities vs. actual classes
ggplot(test_data, aes(x = pred_prob, fill = am)) +
  geom_histogram(binwidth = 0.1, alpha = 0.6) +
  labs(title = "Predicted Probabilities vs. Actual Classes", x = "Predicted Probability", y = "Count") +
  theme_minimal()
# Step 9: Save the model for future use (optional)
saveRDS(logistic_model, "logistic_model.rds")
# Step 10: Load the model and make predictions on new data (optional)
# logistic_model <- readRDS("logistic_model.rds")
# new_data$pred_prob <- predict(logistic_model, new_data, type = "response")
```

GENERALIZED NAÏVE BAYES WITHOUT LIBRARY

```r
# Function to calculate Naïve Bayes probabilities and make a prediction
naive_bayes_predict <- function(data, target, input_values) {
  # Convert all categorical columns to factors
  data[] <- lapply(data, as.factor)
  # Compute prior probabilities P(Y)
  prior_probs <- table(data[[target]]) / nrow(data)
  # Function to compute conditional probability P(X | Y)
  conditional_prob <- function(feature, value, choice) {
    subset_data <- data[data[[target]] == choice, ]  # Filter data for choice Y
    return(sum(subset_data[[feature]] == value) / nrow(subset_data))
  }  # Compute likelihoods P(Feature | Choice) for all input values
  likelihoods <- list()
  for (choice in levels(data[[target]])) {
    likelihoods[[choice]] <- sapply(names(input_values), function(feature) {
      conditional_prob(feature, input_values[[feature]], choice)
    })
  }  # Compute posterior probabilities P(Y | X) using Naïve Bayes formula
  posterior_probs <- sapply(levels(data[[target]]), function(choice) {
    prior_probs[choice] * prod(likelihoods[[choice]])
  })  # Normalize probabilities (optional, since we're just comparing values)
  posterior_probs <- posterior_probs / sum(posterior_probs)
  # Predict the most likely choice
  predicted_choice <- names(which.max(posterior_probs))

  # Print results
  print("Posterior probabilities for each choice:")
  print(posterior_probs)
  cat("\nPredicted category:", predicted_choice, "\n")
  return(predicted_choice)
}# Load dataset (replace with your file)
data <- read.csv("popularkids.csv")
```

```r
# Define the target variable (the category you want to predict)
target <- "Choice"  # Change this based on your dataset
# Define the input values for prediction
input_values <- list(
  "Gender" = "Boy",
  "Grade" = "6",
  "School" = "Elm"
)# Call the function to make predictions
predicted_choice <- naive_bayes_predict(data, target, input_values)
```

GENERALIZED NAÏVE BAYES WITH LIBRARY

```r
# Load necessary libraries
library(e1071)    # Naïve Bayes classifier
library(ggplot2)  # For discretizing numerical variables
library(caret)    # For data splitting
# Generalized Naïve Bayes Function
myBayes <- function(data, target, k) {
  set.seed(12345)  # Reproducibility
    # Convert specified continuous variables to numeric (if not already)
  if (!is.numeric(data$Age)) data$Age <- as.numeric(as.character(data$Age))
  if (!is.numeric(data$Limit_bal)) data$Limit_bal <- as.numeric(as.character(data$Limit_bal))
    # Convert categorical variables to factors (except target)
  categorical_vars <- setdiff(names(data), c("Age", "Limit_bal", target))
  for (var in categorical_vars) {
    if (!is.factor(data[[var]])) data[[var]] <- as.factor(data[[var]])
  }  # Discretize continuous variables into k categories
  data$Age <- cut_interval(data$Age, n = k)
  data$Limit_bal <- cut_interval(data$Limit_bal, n = k)
    # Split data: 40% train, 30% validation, 30% test
  trainIndex <- createDataPartition(data[[target]], p = 0.4, list = FALSE)
  trainData <- data[trainIndex, ]
  tempData <- data[-trainIndex, ]
  validIndex <- createDataPartition(tempData[[target]], p = 0.5, list = FALSE)
  validData <- tempData[validIndex, ]
  testData <- tempData[-validIndex, ]   # Train Naïve Bayes model
  nb_model <- naiveBayes(as.formula(paste(target, "~ .")), data = trainData)
    # Predict and calculate errors
  trainError <- mean(predict(nb_model, trainData) != trainData[[target]])
  validError <- mean(predict(nb_model, validData) != validData[[target]])
  testError <- mean(predict(nb_model, testData) != testData[[target]])    # Return results
  return(list(model = nb_model, trainError = trainError, validError = validError, testError = testError
}# Apply myBayes for k = 2 to 10 and store errors
errors <- data.frame(k = 2:10, trainError = NA, validError = NA, testError = NA)
for (k in 2:10) {
  result <- myBayes(data, target = "default_payment", k)
  errors[errors$k == k, "trainError"] <- result$trainError
  errors[errors$k == k, "validError"] <- result$validError
  errors[errors$k == k, "testError"] <- result$testError
}# Plot training, validation, and test errors
ggplot(errors, aes(x = k)) +
  geom_line(aes(y = trainError, color = "Train Error")) +
  geom_line(aes(y = validError, color = "Validation Error")) +
```

```
  geom_line(aes(y = testError, color = "Test Error")) +
  labs(title = "Error Rate vs Number of Categories (k)",
       x = "Number of Categories (k)",
       y = "Error Rate") +
  scale_color_manual(values = c("red", "blue", "green"))# Find best k (lowest validation error)
best_k <- errors$k[which.min(errors$validError)]
cat("\nBest k:", best_k, "\n")# Train model with best k and analyze feature importance
best_model <- myBayes(data, target = "default_payment", k = best_k)$model
obs1_posterior <- predict(best_model, data[1, ], type = "raw")  # Get probabilities
feature_importance <- best_model$tables  # Check feature importance
cat("\nPredicted probabilities for first observation:\n")
print(obs1_posterior)
cat("\nFeature influence in Naïve Bayes model:\n")
print(feature_importance)
```

DECISION TREE WITH ALL THE PARAMETERS(CV SET TO NULL IF NOT NEEDED)

```
library(rpart)
library(caret)
library(ggplot2)
# Function to load dataset from a given file path
load_data <- function(file_path) {
  data <- read.csv(file_path)
  return(data)
}# Function to preprocess data (handle missing values)
preprocess_data <- function(data) {
  for (col in colnames(data)) {
    if (any(is.na(data[[col]]))) {
      data[[col]][is.na(data[[col]])] <- median(data[[col]], na.rm = TRUE)  # Replace NA with column me
    }
  }
  return(data)
}# Function to train a decision tree with optional cross-validation
train_decision_tree <- function(X_train, y_train, minsplit = 2, min_deviation = 0.0) {
  control <- rpart.control(minsplit = minsplit, cp = min_deviation)  # Set tree parameters
  model <- rpart(y_train ~ ., data = X_train, method = "class", control = control, xval = 10)  # 10-fol
  return(model)
}# Function to calculate the maximum number of leaves in the decision tree
calculate_max_leaves <- function(model) {
  return(sum(model$frame$var == "<leaf>"))
}# Function to plot cross-validation error and determine optimal cp
plot_cv_results <- function(model) {
  plotcp(model)  # Plot cross-validation error vs. complexity parameter
  optimal_cp <- model$cptable[which.min(model$cptable[, "xerror"]), "CP"]
  cat("Optimal CP (complexity parameter):", optimal_cp, "\n")
  return(optimal_cp)
}# Function to evaluate the model (calculate errors and confusion matrix)
evaluate_model <- function(model, X_train, y_train, X_val, y_val) {
  train_pred <- predict(model, X_train, type = "class")
  val_pred <- predict(model, X_val, type = "class")
  train_error <- mean(train_pred != y_train)  # Compute training error
  val_error <- mean(val_pred != y_val)  # Compute validation error
  conf_matrix <- table(y_val, val_pred)  # Generate confusion matrix
```

```
  return(list(train_error = train_error, val_error = val_error, conf_matrix = conf_matrix))
}# Main function to execute the workflow
main <- function() {
  file_path <- "data.csv"  # Modify as needed to specify the correct dataset path
  data <- load_data(file_path)
  data <- preprocess_data(data)
  target_column <- "your_target_column"  # Set the name of the target variable manually
  X <- data[, !names(data) %in% target_column]  # Select all columns except the target
  y <- data[[target_column]]  # Select the target column
  set.seed(42)  # Ensure reproducibility
  train_index <- sample(1:nrow(data), 0.8 * nrow(data))  # 80% training, 20% validation split
  X_train <- X[train_index, ]
  y_train <- y[train_index]
  X_val <- X[-train_index, ]
  y_val <- y[-train_index]
  # Train decision tree with specified parameters
  model <- train_decision_tree(X_train, y_train, minsplit = 4, min_deviation = 0.01)
  # Compute and print the maximum number of leaves in the tree
  max_leaves <- calculate_max_leaves(model)
  print(paste("Max Leaves in Tree:", max_leaves))
    # Plot cross-validation results and determine optimal cp
  optimal_cp <- plot_cv_results(model)
    # Evaluate model performance
  evaluation <- evaluate_model(model, X_train, y_train, X_val, y_val)
  print(paste("Training Error:", evaluation$train_error))
  print(paste("Validation Error:", evaluation$val_error))
  print("Confusion Matrix:")
  print(evaluation$conf_matrix)
}
# Run the main function
main()
```

LOGISTIC REGRESSION(LOSSMATRIX NEED TO BE CHANGED OR SET TO NULL, BOOTSTRAP CAN BE SET TO 0 IS NOT REQURIED)

```
library(caret)  # For confusion matrix and performance metrics
library(boot)   # For bootstrapping
library(readr)  # For reading CSV files
data <- read_csv("your_file.csv")# Load dataset from CSV
# Check dataset structure and first few rows
str(data)
head(data)
# Function to perform logistic regression
logistic_regression <- function(data, dependent_var, independent_vars, loss_matrix = NULL, n_boot = 100
  # Create formula for logistic regression
  formula <- as.formula(paste(dependent_var, "~", paste(independent_vars, collapse = "+")))
  # Split data into train (70%) and test (30%)
  set.seed(123)  # For reproducibility
  trainIndex <- createDataPartition(data[[dependent_var]], p = 0.7, list = FALSE)
  train_data <- data[trainIndex, ]
  test_data  <- data[-trainIndex, ]
  # Fit logistic regression model
  model <- glm(formula, data = train_data, family = binomial)
```

```r
# Display model summary
print(summary(model))
# Function to calculate model coefficients for bootstrapping
boot_fn <- function(data, indices) {
  model <- glm(formula, data = data[indices, ], family = binomial)
  return(coef(model))
}
# Perform parametric bootstrapping
boot_results <- boot(data = train_data, statistic = boot_fn, R = n_boot)
print(boot_results)
# Predict on train and test data
train_pred_prob <- predict(model, train_data, type = "response")
test_pred_prob  <- predict(model, test_data, type = "response")
# Set threshold for classification (0.5 by default)
train_pred <- ifelse(train_pred_prob > 0.5, 1, 0)
test_pred  <- ifelse(test_pred_prob > 0.5, 1, 0)
# Calculate misclassification error
train_error <- mean(train_pred != train_data[[dependent_var]])
test_error  <- mean(test_pred != test_data[[dependent_var]])
cat("Train Misclassification Error:", train_error, "\n")
cat("Test Misclassification Error:", test_error, "\n")
# Confusion matrix
cat("Confusion Matrix - Train:\n")
print(table(Predicted = train_pred, Actual = train_data[[dependent_var]]))
cat("Confusion Matrix - Test:\n")
print(table(Predicted = test_pred, Actual = test_data[[dependent_var]]))
# Function to compute F1-score
compute_f1 <- function(actual, predicted) {
  cm <- table(Predicted = predicted, Actual = actual)
  precision <- ifelse(sum(cm[2, ]) == 0, 0, cm[2, 2] / sum(cm[2, ]))
  recall <- ifelse(sum(cm[, 2]) == 0, 0, cm[2, 2] / sum(cm[, 2]))
  if (precision + recall == 0) {
    return(0)
  } else {
    f1 <- 2 * (precision * recall) / (precision + recall)
    return(f1)
  }
}
# F1-score before applying the loss matrix
cat("F1 Score (Before Loss Matrix) - Train:", compute_f1(train_data[[dependent_var]], train_pred), "\n
cat("F1 Score (Before Loss Matrix) - Test:", compute_f1(test_data[[dependent_var]], test_pred), "\n")
# Apply loss matrix (if provided)
if (!is.null(loss_matrix)) {
  # Apply loss matrix logic
  adjusted_train_pred <- train_pred
  adjusted_test_pred <- test_pred
  # Modify predictions based on the loss matrix
  adjusted_train_pred[train_pred == 0 & train_data[[dependent_var]] == 1] <-
    ifelse(loss_matrix[2, 1] > loss_matrix[1, 2], 1, 0)
  adjusted_test_pred[test_pred == 0 & test_data[[dependent_var]] == 1] <-
    ifelse(loss_matrix[2, 1] > loss_matrix[1, 2], 1, 0)
  # Compute F1-score after applying the loss matrix
  cat("F1 Score (After Loss Matrix) - Train:", compute_f1(train_data[[dependent_var]], adjusted_train
```

```
    cat("F1 Score (After Loss Matrix) - Test:", compute_f1(test_data[[dependent_var]], adjusted_test_pr
  }
}
# Example usage
# data: Your dataset (e.g., mtcars)
# dependent_var: Binary outcome variable (e.g., "am")
# independent_vars: List of predictor variables (e.g., c("hp", "wt"))

# Define the loss matrix
loss_matrix <- matrix(c(0, 1, 10, 0), nrow = 2, byrow = TRUE)
# Example call to the function
logistic_regression(data, "am", c("hp", "wt"), loss_matrix)
```

PCA (WITH AND WITHOUT SCALING, 90-95%VARIENCE)

```
library(ggplot2)
data <- read.csv("your_file.csv", header = TRUE)# Load the dataset from a CSV file
# Remove categorical columns if necessary (PCA requires numeric data)
data_numeric <- data[sapply(data, is.numeric)]
# Function for PCA with different options
perform_pca <- function(data, scale_data = TRUE, variance_threshold = 0.90, max_components = 10) {
  # Perform PCA
  pca_model <- prcomp(data, center = TRUE, scale. = scale_data)
  # Get explained variance
  explained_variance <- pca_model$sdev^2 / sum(pca_model$sdev^2)
  cumulative_variance <- cumsum(explained_variance)
  # Find number of components to retain the required variance
  num_components <- min(which(cumulative_variance >= variance_threshold))
  num_components <- min(num_components, max_components) # Restrict to max_components
  # Trace plot of explained variance
  df_var <- data.frame(Component = 1:length(explained_variance), Variance = cumulative_variance)
  ggplot(df_var, aes(x = Component, y = Variance)) +
    geom_line(color = 'blue') +
    geom_point(color = 'red') +
    geom_hline(yintercept = variance_threshold, linetype = 'dashed', color = 'black') +
    labs(title = 'Cumulative Variance Explained', x = 'Principal Component', y = 'Cumulative Variance')
  # Compute PCA coordinates in original space and PC coordinate system
  pca_coordinates <- predict(pca_model)[, 1:num_components]
  # Convert PCA coordinates to a DataFrame and rename columns
  pc_data <- as.data.frame(pca_coordinates)
  colnames(pc_data) <- paste0("PC", 1:ncol(pc_data))

  # Train-test split (80-20 split)
  set.seed(123)
  sample_indices <- sample(1:nrow(data), 0.8 * nrow(data))
  train_data <- data[sample_indices, ]
  test_data <- data[-sample_indices, ]
  # Train and test error computation
  train_pca <- prcomp(train_data, center = TRUE, scale. = scale_data)
  train_reconstructed <- train_pca$x[, 1:num_components] %*% t(train_pca$rotation[, 1:num_components])
  train_error <- mean((train_data - train_reconstructed)^2)
  test_projected <- scale(test_data, center = train_pca$center, scale = train_pca$scale) %*% train_pca$
  test_reconstructed <- test_projected %*% t(train_pca$rotation[, 1:num_components])
```

```
  test_error <- mean((test_data - test_reconstructed)^2)
  return(list(pca_model = pca_model, num_components = num_components,
              train_error = train_error, test_error = test_error,
              pca_coordinates = pc_data))
}
# Perform PCA on the loaded data
result <- perform_pca(data_numeric, scale_data = TRUE)
# Print results
print(result$num_components)
print(result$train_error)
print(result$test_error)
```

KNN WITH ALL POSSIBLE WAYS

```
library(class)
library(caret)
# Function to calculate MSE
mse <- function(actual, predicted) {
  mean((actual - predicted)^2)
}
# Function to perform KNN and calculate MSE
run_knn <- function(data, target_col, split_ratio = c(0.7, 0.3), k_values = 1:99, specific_ks = c(5, 10
  set.seed(123)  # For reproducibility
  # Splitting data into 2 or 3 parts
  indices <- createDataPartition(data[[target_col]], p = split_ratio[1], list = FALSE)
  train_data <- data[indices, ]
  remaining_data <- data[-indices, ]
  if (length(split_ratio) == 3) {
    indices2 <- createDataPartition(remaining_data[[target_col]], p = split_ratio[2] / sum(split_ratio[1
    test_data <- remaining_data[indices2, ]
    validation_data <- remaining_data[-indices2, ]
  } else {
    test_data <- remaining_data
    validation_data <- NULL
  }
  # Preparing predictor and response variables
  train_x <- train_data[, !names(train_data) %in% target_col]
  train_y <- train_data[[target_col]]
  test_x <- test_data[, !names(test_data) %in% target_col]
  test_y <- test_data[[target_col]]
  # KNN Loop for k = 1 to 99
  mse_results <- data.frame(K = integer(), Train_MSE = numeric(), Test_MSE = numeric())
  for (k in k_values) {
    pred_train <- knn(train_x, train_x, train_y, k = k)
    pred_test <- knn(train_x, test_x, train_y, k = k)
    train_mse <- mse(as.numeric(as.character(train_y)), as.numeric(as.character(pred_train)))
    test_mse <- mse(as.numeric(as.character(test_y)), as.numeric(as.character(pred_test)))
    mse_results <- rbind(mse_results, data.frame(K = k, Train_MSE = train_mse, Test_MSE = test_mse))
  }
  # KNN for specific values of K
  specific_results <- data.frame(K = integer(), Train_MSE = numeric(), Test_MSE = numeric())
  for (k in specific_ks) {
    pred_train <- knn(train_x, train_x, train_y, k = k)
```

```
    pred_test <- knn(train_x, test_x, train_y, k = k)
    train_mse <- mse(as.numeric(as.character(train_y)), as.numeric(as.character(pred_train)))
    test_mse <- mse(as.numeric(as.character(test_y)), as.numeric(as.character(pred_test)))
    specific_results <- rbind(specific_results, data.frame(K = k, Train_MSE = train_mse, Test_MSE = tes
  }
  return(list(All_K_MSE = mse_results, Specific_K_MSE = specific_results, Validation_Data = validation_
}# Loading a CSV file
my_data <- read.csv("path/to/your/file.csv")  # Provide the correct path to your CSV file
# Specify the target column
target_column <- "target"  # Replace "target" with the actual column name in your CSV
# Run KNN with the loaded data
results <- run_knn(my_data, target_column, split_ratio = c(0.7, 0.3), k_values = 1:99, specific_ks = c(
# Print the results
print(results$All_K_MSE)
print(results$Specific_K_MSE)
```

LASSO REGRESSION(SCALING, FAMILY, CV,)

```
library(glmnet)  # For LASSO regression
library(caret)   # For data splitting
set.seed(123) # Set seed for reproducibility
# Load your dataset (Replace with actual file or dataset)
data <- read.csv("your_dataset.csv")
# Define target (y) and predictors (X)
y <- data$target_variable  # Replace with actual target column name
X <- as.matrix(data[, !names(data) %in% "target_variable"])  # Exclude target from predictors
# Split data into training (80%) and test (20%) sets
train_index <- createDataPartition(y, p = 0.8, list = FALSE)
X_train <- X[train_index, ]
y_train <- y[train_index]
X_test <- X[-train_index, ]
y_test <- y[-train_index]
# ---- USER CHOICES ----
scale_features <- TRUE  # Set to FALSE to disable scaling
distribution_family <- "gaussian"  # Change to "poisson", "binomial", etc. based on your target
# ---- FEATURE SCALING (if enabled) ----
if (scale_features) {
  X_train <- scale(X_train)
  X_test <- scale(X_test, center = attr(X_train, "scaled:center"),
                  scale = attr(X_train, "scaled:scale"))
}
# ---- FIT LASSO MODEL ----
lasso_model <- glmnet(X_train, y_train, alpha = 1, family = distribution_family, standardize = FALSE)
# "standardize = FALSE" ensures glmnet does not scale internally (we handle it manually)
# ---- PERFORM CROSS-VALIDATION TO FIND OPTIMAL LAMBDA ----
cv_model <- cv.glmnet(X_train, y_train, alpha = 1, family = distribution_family, standardize = FALSE)
# Extract best lambda values
lambda_min <- cv_model$lambda.min  # Best fit (min error)
lambda_1se <- cv_model$lambda.1se  # Simpler model (within 1 standard error)
# Print lambda values
cat("Lambda.min (best fit):", lambda_min, "\n")
cat("Lambda.1se (simpler model):", lambda_1se, "\n")
# ---- PLOT CROSS-VALIDATION RESULTS ----
```

```r
plot(cv_model)
abline(v = log(lambda_min), col = "blue", lty = 2)   # Lambda.min
abline(v = log(lambda_1se), col = "red", lty = 2)    # Lambda.1se
# ---- EXTRACT SELECTED FEATURES ----
coefs_min <- coef(cv_model, s = "lambda.min")  # Coefficients at lambda.min
coefs_1se <- coef(cv_model, s = "lambda.1se")  # Coefficients at lambda.1se
# Identify non-zero coefficients (selected features)
selected_features_min <- rownames(coefs_min)[coefs_min[,1] != 0]
selected_features_1se <- rownames(coefs_1se)[coefs_1se[,1] != 0]
# Count number of selected features (excluding intercept)
num_features_min <- length(selected_features_min) - 1
num_features_1se <- length(selected_features_1se) - 1
# Print selected features
cat("\nSelected Features using Lambda.min (best fit):", num_features_min, "features\n", selected_feature
cat("\nSelected Features using Lambda.1se (simpler model):", num_features_1se, "features\n", selected_fe
# ---- MAKE PREDICTIONS USING BOTH LAMBDAS ----
y_train_pred_min <- predict(cv_model, newx = X_train, s = "lambda.min")
y_test_pred_min <- predict(cv_model, newx = X_test, s = "lambda.min")
y_train_pred_1se <- predict(cv_model, newx = X_train, s = "lambda.1se")
y_test_pred_1se <- predict(cv_model, newx = X_test, s = "lambda.1se")
# ---- COMPUTE MEAN SQUARED ERROR (MSE) ----
mse_train_min <- mean((y_train - y_train_pred_min)^2)
mse_test_min <- mean((y_test - y_test_pred_min)^2)
mse_train_1se <- mean((y_train - y_train_pred_1se)^2)
mse_test_1se <- mean((y_test - y_test_pred_1se)^2)
# ---- PRINT MSE RESULTS ----
cat("\nMSE using Lambda.min:\n")
cat("Training MSE:", mse_train_min, "\n")
cat("Test MSE:", mse_test_min, "\n")
cat("\nMSE using Lambda.1se:\n")
cat("Training MSE:", mse_train_1se, "\n")
cat("Test MSE:", mse_test_1se, "\n")
```

OPTIMIZATION

```r
# library(ggplot2) # Load required packages
# Function to calculate the loss (customizable)
calculate_loss <- function(params, X, y, loss_type = "L1") {
  y_hat <- as.matrix(X) %*% params
  if (loss_type == "L1") {
    return(sum(abs(y - y_hat)))
  } else if (loss_type == "L2") {
    return(sum((y - y_hat)^2))
  } else if (loss_type == "log_likelihood") {
    sigma2 <- 0.1 * X[, 1] - 0.5
    return(-sum(dnorm(y, mean = y_hat, sd = sqrt(sigma2), log = TRUE)))
  } else if (loss_type == "exp_loss") {
    return(sum(exp(-y * y_hat)))
  } else {
    stop("Invalid loss type")
  }
}# Optimization function
optimize_model <- function(X, y, init_params, method = "BFGS", loss_type = "L1", max_iter = 100) {
```

```
  optim_result <- optim(
    par = init_params,
    fn = calculate_loss,
    X = X, y = y, loss_type = loss_type,
    method = method,
    control = list(maxit = max_iter, trace = TRUE)
  )
  return(optim_result)
}# Function to plot error vs iterations
plot_error_vs_iterations <- function(errors) {
  df <- data.frame(iter = seq_along(errors), error = errors)
  ggplot(df, aes(x = iter, y = error)) +
    geom_line(color = "blue") +
    labs(title = "Error vs Iterations", x = "Iteration", y = "Error")
}# Scatter plot function
plot_scatter <- function(X, y, title) {
  df <- data.frame(Humidity = X[, 1], Visibility = X[, 2], Target = y)
  ggplot(df, aes(x = Humidity, y = Visibility, color = Target)) +
    geom_point() +
    labs(title = title)
}# Main workflow
set.seed(123)# Load dataset (replace with your actual dataset)
my_data <- read.csv("your_data.csv")
# Define input (X) and target (y)
X <- as.matrix(my_data[, c("Humidity", "Visibility")])  # Choose your features
y <- my_data$DewPointTemperature  # Set your target variable
# Split data into training and test (50/50)
train_indices <- sample(1:nrow(X), size = 0.5 * nrow(X))
X_train <- X[train_indices, ]
y_train <- y[train_indices]
X_test <- X[-train_indices, ]
y_test <- y[-train_indices]
# Initial parameters and optimization
init_params <- rep(0, ncol(X_train))
result <- optimize_model(X_train, y_train, init_params, method = "BFGS", loss_type = "L1", max_iter = 2
print(result)
plot_error_vs_iterations(result$counts) # Plotting
plot_scatter(X_train, y_train, "Original Training Data")
plot_scatter(X_train, X_train %*% result$par, "Predicted Training Data")
plot_scatter(X_test, X_test %*% result$par, "Predicted Test Data")
```

RIDGE CLASSIFICATION

```
library(caret)  # For data partitioning, confusion matrix, etc.
library(glmnet) # For Ridge regression (glmnet)
library(tidyverse)  # For data manipulation (optional)
# Load your dataset (modify the path or dataset name here)
# data <- read.csv("path/to/your/data.csv")  # If your data is in a CSV file
set.seed(123)  # For reproducibility# Split the data into training and test sets (50/50)
trainIndex <- createDataPartition(data$Survival, p = 0.5, list = FALSE)  # Modify 'Survival' if needed
trainData <- data[trainIndex, ]
testData <- data[-trainIndex, ]
# Specify the feature matrix (X) and target variable (Y)
```

```r
# Modify 'Survival' and ensure all feature columns are included
x_train <- trainData %>% select(-Survival)  # All columns except 'Survival'
y_train <- trainData$Survival  # 'Survival' is the target variable
x_test <- testData %>% select(-Survival)  # Modify based on your target variable
y_test <- testData$Survival
# Train the Ridge classifier using cross-validation to select the optimal penalty parameter (lambda)
# alpha = 0 specifies Ridge regression, lambda will be optimized by cross-validation
ridge_model <- cv.glmnet(as.matrix(x_train), y_train, alpha = 0, family = "binomial",
                         type.measure = "class", nfolds = 10)  # Modify nfolds or type.measure if needed
# Report the optimal penalty parameter (lambda)
optimal_lambda <- ridge_model$lambda.min
cat("Optimal Penalty Parameter (Lambda):", optimal_lambda, "\n")
# Fit the Ridge model using the optimal lambda
ridge_final_model <- glmnet(as.matrix(x_train), y_train, alpha = 0, lambda = optimal_lambda, family = "
predictions <- predict(ridge_final_model, newx = as.matrix(x_test), type = "class")
conf_matrix <- confusionMatrix(predictions, y_test) # Generate the confusion matrix for the test set
print(conf_matrix)
```

ml Lab

Varun Gurupurandar

2025-06-23

KERNEL MODELS:Kernel models can also be used for density estimation, i.e., to model a probability distribution or density function p(x). p(x*)=1/n sum from i=1 to n K((x*-xi)/h).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.The kernel model presented above can be used to estimate the class conditional density functions p(x|class=1) and p(x|class=2). These can in their turn be used to produce posterior class probabilities p(class|x) via Bayes theorem. Specifically, p(class=1|x) = p(x|class=1) p(class=1) / [p(x|class=1) p(class=1) + p(x|class=2) p(class=2)] You are asked to come up with a dataset of your own to illustrate the effect of h in the generalization error. Specifically, you should show that overfitting occurs for certain value of h, and underfitting occurs for some other value of h. ALTER: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, . . . , 4.9, 5.

```
set.seed(123456789)
N_class1 <- 1000#VALUE GIVEN CHANGE IF NEEDED
N_class2 <- 1000#VALUE GIVEN CHANGE IF NEEDED
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)
}
# Estimate the class conditional density functions: 2p.
conditional_class1 <- function(t, h){
 d <- 0
 for(i in 1:800)#VALUE GIVEN CHANGE IF NEEDED
   d <- d+dnorm((t-data_class1[i])/h)
 return (d/800)
}
conditional_class2 <- function(t, h){
 d <- 0
 for(i in 1:800)#VALUE GIVEN CHANGE IF NEEDED
   d <- d+dnorm((t-data_class2[i])/h)
 return (d/800)
}
```

```r
# Estimate the class posterior probability distribution: 1p.
prob_class1 <- function(t, h){
 prob_class1 <- conditional_class1(t,h)*800/1600
 prob_class2 <-conditional_class2(t,h)*800/1600
 return (prob_class1/(prob_class1 + prob_class2))
}
# Select h value via validation: 2p.
foo <- NULL
for(h in seq(0.1,5,0.1)){
 foo <- c(foo, (sum(prob_class1(data_class1[801:900], h)>0.5)+
                  sum(prob_class1(data_class2[801:900], h)<0.5))/200)#VALUE GIVEN CHANGE IF NEEDED
}
plot(seq(0.1,5,0.1),foo)
max(foo)
which(foo==max(foo))*0.1
# Estimate the generalization error: 1p.
# To estimate the generalization error,we use the best h value found previously.
#that the training data is now the old training data union the validation data.
#Using just the old training data results results in an estimate that is a bittoo pessimistic.
conditional_class1 <- function(t, h){
 d <- 0
 for(i in 1:900)#VALUE GIVEN CHANGE IF NEEDED
   d <- d+dnorm((t-data_class1[i])/h)
 return (d/900)
}
conditional_class2 <- function(t, h){
 d <- 0
 for(i in 1:900)#VALUE GIVEN CHANGE IF NEEDED
   d <- d+dnorm((t-data_class2[i])/h)

 return (d/900)
}
prob_class1 <- function(t, h){
 prob_class1 <- conditional_class1(t,h)*900/1800
 prob_class2 <-conditional_class2(t,h)*900/1800
 return (prob_class1/(prob_class1 + prob_class2))
}
h <- which(foo==max(foo))*0.1
(sum(prob_class1(data_class1[901:1000], h)>0.5)+sum(prob_class1(data_class2[901:1000], h)<0.5))/200
#VALUE GIVEN CHANGE IF NEEDED
```

NEURAL NETWORKS:train a neural network for substracting two numbers from the interval [-1,1]. Look at the plot of the learned neural network and explain why the weights learned make sense.

```r
library(neuralnet)
set.seed(1234567890)
x1 <- runif(1000, -1, 1)
x2 <- runif(1000, -1, 1)
tr <- data.frame(x1,x2, y=x1 - x2)
winit <- runif(9, -1, 1)
nn<-neuralnet(formula = y ~ x1 + x2, data = tr, hidden = c(1), act.fct = "tanh")
plot(nn)
#The neural network successfully learns to subtract two numbers, \( x_1 \) and \( x_2 \),
#in the range \([-1, 1]\) because the problem is linear, and the network's structure is simple yet effe
```

```
#The ideal solution requires the weights to encode the subtraction operation \( y = x_1 - x_2 \),
#which the network achieves by assigning a weight close to \( +1 \) for \( x_1 \) and \( -1 \)
#for \( x_2 \)in the hidden neuron. The output neuron then uses a weight near \( +1 \) to preserve the
#while biasesremain close to 0 since no offset is needed.The \(\tanh\) activation function works well h
#because, forsmall input values, it behaves almost linearly, meaning \( \tanh(x) \approx x \). As a res
#the hiddenneuron approximates \( \tanh(x_1 - x_2) \approx x_1 - x_2 \), allowing the network to produc
#accurate subtractions.The learned weights align perfectly with the mathematical operation, demonstrati
#that even a minimal neural networkcan solve linear problems efficiently when the
#activation function and architecture areell-suited.This example highlights how neural networks can mod
#operations by learning the correct weight configurations.
```

NEURAL NETWORKS: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.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:1. Divide the learning data into approximately equal sized folds D1 and D2->2. Train the regressor on D1 and test it on D2->3.Train the regressor on D2 and test it on D1->4.Report the average error on the test folds. . . answer to the following question:What is the NN that you should return to the user?The one learned from D1? The one learned from D2? Either of them? None?. . . . . .name one advantage and one disadvantage of using cross-validation to estimate the generalization error.

```
library(neuralnet)
set.seed(1234567890)
Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr1 <- trva[1:25,]
tr2 <- trva[26:50,]
# Estimate the generalization error: 3p
winit <- runif(31, -1, 1)
nn1 <- neuralnet(formula = Sin ~ Var, data = tr1, hidden = 10, startweights = winit,
                 threshold = 0.001, lifesign = "full")
winit <- runif(31, -1, 1)
nn2 <- neuralnet(formula = Sin ~ Var, data = tr2, hidden = 10, startweights = winit,
                threshold = 0.001, lifesign = "full")
aux1 <- predict(nn1, tr2)
aux2 <- predict(nn2, tr1)
sum((tr2[,2] - aux1)**2)/2 + sum((tr1[,2] - aux2)**2)/2
# NN to return to the user: 1p
winit <- runif(31, -1, 1)
nn1 <- neuralnet(formula = Sin ~ Var, data = rbind(tr1,tr2), hidden = 10, startweights = winit,
                 threshold = 0.001, lifesign = "full")
#In the given task, the objective is to train a neural network (NN) to approximate the sine function
#using data sampled uniformly from the interval $[0, 10]$, and to estimate its generalization
#error using 2-fold cross-validation. During cross-validation, the dataset is split into two equal part
#(D1 and D2).The model is trained on D1 and tested on D2, and then trained on D2 and tested on D1. The
#average testerror from both folds provides an estimate of the NN's generalization performance. However
#the purpose of cross-validation is solely to assess how well a model is likely to generalize to unseen
#data. It is notintended to identify a single "best" model among the folds. Therefore, after
#evaluating the generalization error, the correct approach is to train a final neural network on the
#**entire dataset**, combiningboth D1 and D2.This way, the final model benefits from all
#available training data, leading to bettergeneralization and predictive accuracy.
```

```
#Returning either of the networks trained on only D1 or D2 would be suboptimal,as each has
#seen only half the data. Thus, the neural network that should be returned to
#the user is the onetrained on the **full dataset** after cross-validation.

#Crossvalidation is reliable method for estimating model's generalization error.Its
#main advantage is that it provides a more accurate and stable estimate by training
#testing the model on different subsets of data.In 2-fold cross-validation,data is split
#into two equal parts;model is trained on one and tested on other,then roles are reversed.
#The test errors are averaged to estimate performance on unseen data.A key disadvantage is
#increased computational cost,as model must be trained multiple times, which can be time
#consuming for complex models orlarge datasets.
```

NEURAL NETWORKS:You are asked to implement the backpropagation algorithm for training a neural network for regression as it appears in the course textbook and slides. You can find the pseudocode below. The neural network has one hidden layer with two units. W denotes weights,b denotes intercepts,z denotes activation units,q denotes hidden units,J denotes the squared error, and gamma denotes the learning rate.The superscript indicates the layer(0=input layer,1=hidden layer,2=output layer).All products are matrixproducts (% * %),except the one indicated with that is element-wise product (*).the algorithm performs 100000 iterations In each iteration, one randomly selected training point is used to update the parameters(thiscorresponds to stochastic gradient descent withmini-batch size 1). asked to incorporate dropout to your implementation of the backpropagation algorithm.Recall that dropout is a regularization technique Run implementation with a dropout rate 1-r equal to 0, 0.01 and 0.05, i.e. r=1, 0.99, 0.95.

```
set.seed(123)
# Generate training data
x <- runif(500, -4, 4)
y <- sin(x)
dat <- cbind(x, y)
# Activation and derivative (sigmoid)
h <- function(z) 1 / (1 + exp(-z))
hprime <- function(z) h(z) * (1 - h(z))
# Training function
train_net <- function(dat, gamma = 0.01, steps = 100000, r = 1) {
  # Initialize weights and biases
  W1 <- matrix(rnorm(2), 2, 1); b1 <- matrix(0, 2, 1)
  W2 <- matrix(rnorm(2), 1, 2); b2 <- 0
  # Prediction function
  yhat <- function(xi) {
    z1 <- W1 %*% xi + b1
    q1 <- h(z1)
    return(as.numeric(W2 %*% q1 + b2))  # No dropout at test time
  }
  # MSE
  MSE <- function() {
    pred <- sapply(dat[,1], yhat)
    mean((pred - dat[,2])^2)
  }
  res <- NULL
  for (i in 1:steps) {
    if (i %% 1000 == 0) res <- c(res, MSE())
    j <- sample(1:nrow(dat), 1)
    xj <- dat[j, 1]; yj <- dat[j, 2]
    z1 <- W1 %*% xj + b1
```

```
    q1 <- h(z1)
    D <- matrix(rbinom(2, 1, r), 2, 1)
    q1d <- q1 * D / r
    z2 <- W2 %*% q1d + b2
    y_pred <- as.numeric(z2)
    dz2 <- 2 * (y_pred - yj)
    dW2 <- dz2 * t(q1d); db2 <- dz2
    dz1 <- t(W2) %*% dz2 * hprime(z1) * D / r
    dW1 <- dz1 %*% t(xj); db1 <- dz1
    W2 <- W2 - gamma * dW2; b2 <- b2 - gamma * db2
    W1 <- W1 - gamma * dW1; b1 <- b1 - gamma * db1
  }
  list(MSE = res, predict = function(xvals) sapply(xvals, yhat))
}
# Run training with different dropout rates
model_r1    <- train_net(dat, r = 1)
model_r099  <- train_net(dat, r = 0.99)
model_r095  <- train_net(dat, r = 0.95)
# Plot training error
plot(model_r1$MSE, type = "l", col = "blue", ylim = range(c(model_r1$MSE,
                                                            model_r099$MSE, model_r095$MSE)),
     xlab = "Iterations (x1000)", ylab = "MSE", main = "Training Error for Different Dropout Rates")
lines(model_r099$MSE, col = "green")
lines(model_r095$MSE, col = "red")
legend("topright", legend = c("Dropout 0%", "Dropout 1%", "Dropout 5%"),
       col = c("blue", "green", "red"), lty = 1)
# Plot predictions
plot(dat, main = "Predictions vs True Function")
points(dat[,1], model_r1$predict(dat[,1]), col = "blue", pch = 20)
points(dat[,1], model_r099$predict(dat[,1]), col = "green", pch = 20)
points(dat[,1], model_r095$predict(dat[,1]), col = "red", pch = 20)
legend("topright", legend = c("Dropout 0%", "Dropout 1%", "Dropout 5%"),
       col = c("blue", "green", "red"), pch = 20)
#------------------WITH DROPOUT -----------------------------
#------------------------------------------------------------------------
set.seed(1234)
# produce the training data in dat
x <- runif(500,-4,4)
y <- sin(x)
dat <- cbind(x,y)
plot(dat)
gamma <- 0.01
r <- 1 ### dropout rate 1-r
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)))
}
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)
}
yhat2 <- function(x){
  ### final prediction for point x (it involves r)
  q0 <- x
  z1 <- (r*w1) %*% q0 + b1
  q1 <- as.matrix(apply(z1,1,h), nrow = 2, ncol = 1)
  z2 <- (r*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))
}# initialize parameters
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)
res <- NULL
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]
  m0 <- sample(c(0,1),1,replace = TRUE, c(1-r,r)) ### drop input unit
  z1 <- w1 %*% (q0*m0) + b1
  q1 <- as.matrix(apply(z1,1,h), nrow = 2, ncol = 1)
  m1 <- sample(c(0,1),2,replace = TRUE, c(1-r,r)) ### drop hidden units
  z2 <- w2 %*% (q1*m1) + b2
  # backward propagation
  dz2 <- - 2 * (dat[j,2] - z2)
  dq1 <- t(w2) %*% dz2
  dz1 <- dq1 * hprime(z1)
  dw2 <- dz2 %*% t(q1)
  db2 <- dz2
  dw1 <- dz1 %*% t(q0)
  db1 <- dz1
  # parameter updating
  w2 <- w2 - (gamma * dw2 * m1) ### update non-dropped hidden units
  b2 <- b2 - gamma * db2
  w1 <- w1 - (gamma * dw1 * c(m0,m0) * m1) ### update non-dropped hidden units
  b1 <- b1 - gamma * db1
}
plot(res, type = "l")
```

```r
plot(dat)
points(dat[,1],lapply(dat[,1],yhat2),col="red") ### apply final prediction
# Lower r value implies larger regularization and, thus, worse fit of the training data.
#The ultimate goal of dropout is to get a better fit of the test data
#(i.e., low generalization error). However, this is too simple an example to observe it.
#----------------WITHOUT DROPOUT-------------------------------
#--------------------------------------------------------------
set.seed(1234)# produce the training data in dat
x <- runif(500,-4,4)
y <- sin(x)
dat <- cbind(x,y)
plot(dat)
gamma <- 0.01
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)))
}
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))
}# initialize parameters
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)
res <- NULL
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)
  z2 <- w2 %*% q1 + b2
  # backward propagation
  dz2 <- - 2 * (dat[j,2] - z2)
  dq1 <- t(w2) %*% dz2
```

```
  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
}
plot(res, type = "l")
plot(dat)
points(dat[,1],lapply(dat[,1],yhat),col="red")
```

Support Vector Machines:Your task is to implement the so-called budget online SVM classifier. 1. S=PHI, b=0 2. Forever: a. Get (x_i,t_i) b. y_i = SUM_{m belongs S} a_m t_m k(x_i,x_m) + b c. If t_i y_i <=BETA: i. S += {i}, a_i=1 ii. If |S|>M: S -= {argmax} t_m(y_m - a_m t_m k(x_m,x_m))} Run your code on the spambase.csv file for the (M, beta) values (500,0) and (500,-0.05).answer the following two questions.explain why the removal criterion in step 8 of the budget online SVM makes sense.name two similarities between the budget online SVM and the SVMs that you have seen in the course

```
set.seed(1234567890)
set.seed(1234567890)
spam <- read.csv2("spambase.csv")
ind <- sample(1:nrow(spam))
spam <- spam[ind, c(1:48, 58)]
h <- 1
beta <- 0  # First case, we'll change to -0.05 for second run
M <- 500
N <- 500# Gaussian kernel function
gaussian_k <- function(x, h) {
  exp(-x^2/(2*h^2))
}# SVM prediction function
SVM <- function(sv, i) {
  t_m <- 2*spam[sv, ncol(spam)] - 1  # Convert labels to -1/+1
  x_i <- spam[i, -ncol(spam)]
  x_m <- spam[sv, -ncol(spam)]
  distances <- sqrt(rowSums((x_m - matrix(x_i, nrow = length(sv), ncol = ncol(x_m),
                                         byrow = TRUE)^2))
  sum(gaussian_k(distances, h) * t_m) + b
}# Initialize variables for first run (beta = 0)
errors <- 1
errorrate <- vector(length = N)
errorrate[1] <- 1
sv <- c(1)
a <- c(1)
b <- 0
# Main training loop for beta = 0
for(i in 2:N) {
  t_i <- 2*spam[i, ncol(spam)] - 1
  y_i <- if(length(sv) > 0) SVM(sv, i) else b
  if(t_i * y_i <= beta) {
```

```
    sv <- c(sv, i)
    a <- c(a, 1)
    if(length(sv) > M) {
      scores <- sapply(1:length(sv), function(m) {
        t_m <- 2*spam[sv[m], ncol(spam)] - 1
        y_m <- if(length(sv[-m]) > 0) SVM(sv[-m], sv[m]) else b
        t_m * (y_m - a[m] * t_m * gaussian_k(0, h))
      })
      sv <- sv[-which.max(scores)]
      a <- a[-which.max(scores)]
    }
  }
  errorrate[i] <- errorrate[i-1] * (i-1)/i + (sign(y_i) != t_i)/i
}
results1 <- list(errorrate = errorrate, sv_count = length(sv))
# Reset variables for second run (beta = -0.05)
beta <- -0.05
errors <- 1
errorrate <- vector(length = N)
errorrate[1] <- 1
sv <- c(1)
a <- c(1)
b <- 0
# Main training loop for beta = -0.05
for(i in 2:N) {
  t_i <- 2*spam[i, ncol(spam)] - 1
  y_i <- if(length(sv) > 0) SVM(sv, i) else b
  if(t_i * y_i <= beta) {
    sv <- c(sv, i)
    a <- c(a, 1)
    if(length(sv) > M) {
      scores <- sapply(1:length(sv), function(m) {
        t_m <- 2*spam[sv[m], ncol(spam)] - 1
        y_m <- if(length(sv[-m]) > 0) SVM(sv[-m], sv[m]) else b
        t_m * (y_m - a[m] * t_m * gaussian_k(0, h))
      })
      sv <- sv[-which.max(scores)]
      a <- a[-which.max(scores)]
    }
  }
  errorrate[i] <- errorrate[i-1] * (i-1)/i + (sign(y_i) != t_i)/i
}
results2 <- list(errorrate = errorrate, sv_count = length(sv))
# Plot results
plot(results1$errorrate[seq(1, N, 10)], ylim = c(0.2, 0.4), type = "o", col = "blue",
     xlab = "Iteration (every 10th)", ylab = "Error Rate", main = "Online SVM Performance")
lines(results2$errorrate[seq(1, N, 10)], type = "o", col = "red")
legend("topright", legend = c("BETA=0", "BETA=-0.05"), col = c("blue", "red"), lty = 1)
# Output results
cat("BETA=0: Final error rate =", round(results1$errorrate[N], 4),
    "| Support vectors =", results1$sv_count, "\n")
cat("BETA=-0.05: Final error rate =", round(results2$errorrate[N], 4),
    "| Support vectors =", results2$sv_count, "\n")# Answers to questions
```

```r
cat("\n1. Removal Criterion Explanation:
The algorithm removes the support vector that contributes least to maintaining the margin.
Specifically, it removes the point m that maximizes t_m(y(x_m) - a_m t_m k(x_m,x_m)),
which represents the point whose removal would affect the decision boundary the least.
This ensures we maintain the most informative support vectors within our budget M.\n")
cat("\n2. Similarities with Batch SVM:
a) Both use kernel functions to handle non-linear decision boundaries
b) Both aim to maximize the classification margin, though online SVM does this incrementally
c) Both rely on support vectors to define the decision boundary\n")
```

You asked implement perceptron algorithm.algorithm for binary classification predecessor modern neural networks.Binary classification with class label t{-1,+1} ,y(w,x)=+1 if w_t x>=0„-1 if w_t x<0. the value of w are iterative determined with the help of data{(x1,t1)..xn,tn} wi=wi+alpha sum(t-y(w-xn))xn.use an alpha value of 0.0001. Plot the the misclassification rate on the dataset below as a function of the number of iterations

```r
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)
t <- ifelse(x[,2]<(x[,1])^2,-1,1)
plot(x[,1],x[,2],col=t+2)
w <- c(0,0)
alpha <- 0.0001
res <- NULL
for(i in 1:100){
  res <- c(res,sum(t != (2 * ((w %*% t(x[,1:2]))>0) - 1))) # note that t is both class label and transp
  w[1] <- w[1] + alpha * sum((t - w %*% t(x[,1:2])) * x[,1])
  w[2] <- w[2] + alpha * sum((t - w %*% t(x[,1:2])) * x[,2])
}
plot(x[,1],x[,2],col=(2 * ((w %*% t(x[,1:2]))>0) - 1)+2)
plot(res, type = "l")
#It works for linearly separable datasets.If the true label islarger than the prediction,then the weigh
#increased for positive inputvaluesanddecreased for negativeinputvalues.Theopposite whentrue label is
#smaller.Explanation:The perceptron algorithm works best when the dataset is linearly separable,
#meaning there exists a straight line that separates the two classes.In those cases,the
#algorithm will converge to a solution that perfectly classifiesall the training data.
#In this example,the true decision boundary is a curve (x2 = x1^2), so the classes are
#NOT linearly separable.This means the perceptron will not fully converge,but it will still try to
#minimize the number of misclassified points.The plot of misclassification rate shows how the
#algorithm improves the decisionboundary over iterations, even if it cannot make it perfect due to
#the curved boundary.Therefore,perceptron works well for linearly separable data, but struggles
#or failsto fully separate non-linear patterns like in this case.
```

Support Vector Machines:function ksvm from the R package kernlab learn support vector machine classifying the spam dataset that included with the package.Consider radial basis function kernel (also known as Gaussian) width of 0.05.For C parameter,consider values 0.5, 1 and 5.This implies that you have to consider three models.Perform model selection,select most promising of the three models (use any method of your choice).Estimate generalization error of SVM selected above.Produce SVM that will be returned to user, show code. What is the purpose of parameter C ?

```r
library(kernlab)# Load required library
data(spam)# Load the spam dataset
set.seed(123)# Set seed for reproducibility
# Split data into training (80%) and testing (20%) for final evaluation
train_idx <- sample(1:nrow(spam), 0.8 * nrow(spam))
train_data <- spam[train_idx, ]
test_data <- spam[-train_idx, ]
rbf_sigma <- 0.05# Set RBF (Gaussian) kernel width
C_values <- c(0.5, 1, 5)# Define candidate C values
# Purpose of parameter C:
# C controls the trade-off between maximizing the margin and minimizing the classification error.
# - Smaller C → More regularization → Wider margin, allows some misclassifications → Better generalizat
# - Larger C → Less regularization → Tries to classify all training data correctly → Can overfit
# In short: C helps control model complexity and the overfitting vs underfitting balance.
# Perform 10-fold cross-validation for each C value
cv_errors <- numeric(length(C_values))
for (i in seq_along(C_values)) {
  model <- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
                kpar = list(sigma = rbf_sigma), C = C_values[i],
                cross = 10)
  cv_errors[i] <- 1 - model@cross  # cross is the average accuracy
}
# Select best C based on lowest cross-validation error
best_C <- C_values[which.min(cv_errors)]
cat("Best C:", best_C, "\n")
# Train final model on full training set using best C
final_model <- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
                    kpar = list(sigma = rbf_sigma), C = best_C)
# Predict on test set and compute generalization error
predictions <- predict(final_model, test_data[,-58])
test_error <- mean(predictions != test_data$type)
cat("Test error (generalization error):", round(test_error, 4), "\n")
# Return the trained final SVM model
final_model
```

Kernel Methods:implementing a kernel method to predict.Modify lab solution classify the spam dataset used in previous exercise.Use Gaussian kernel and show results for different kernel widths,e.g. use 2/3 of data for learning and 1/3 testing. Use only first 48 attributes in the dataset, in addition to last one which is class label. Assume that class label is a continuous random variable

```r
library(kernlab)
data(spam)
spam_data <- spam[, c(1:48, 58)]
# Convert class label to numeric (0 = nonspam, 1 = spam)
spam_data$type <- ifelse(spam_data$type == "spam", 1, 0)
# Set seed for reproducibility
set.seed(123)
# Split data: 2/3 training, 1/3 testing
n <- nrow(spam_data)
train_indices <- sample(1:n, size = round(2 * n / 3))
train_data <- spam_data[train_indices, ]
test_data <- spam_data[-train_indices, ]
# Extract features and target
```

```
x_train <- train_data[, 1:48]
y_train <- train_data[, 49]
x_test <- test_data[, 1:48]
y_test <- test_data[, 49]
# Function to compute RBF kernel SVM regression and evaluate performance
run_model <- function(sigma_value) {
  model <- ksvm(
    x = as.matrix(x_train),
    y = y_train,
    type = "eps-svr",                # Support Vector Regression
    kernel = "rbfdot",               # Gaussian kernel
    kpar = list(sigma = sigma_value),
    C = 1
  )
    # Predict on test set
  y_pred <- predict(model, as.matrix(x_test))
    # Calculate Mean Squared Error (MSE)
  mse <- mean((y_test - y_pred)^2)
  return(list(sigma = sigma_value, MSE = mse))
}
# Try different sigma values (kernel width)
sigma_values <- c(0.001, 0.01, 0.1, 1, 5)
results <- lapply(sigma_values, run_model)
# Display results
cat("Gaussian Kernel Regression Results:\n")
for (res in results) {
  cat(sprintf("Sigma = %.3f  -->  Test MSE = %.4f\n", res$sigma, res$MSE))
}
```

SUPPORT VECTOR MACHINES:code below trains a support vector machine for classification.The problem consists of two continuous inputs,one binary target and 1000 training points.the problem may seem rather easy.the SVM does not perform great.Explain why

```
library(kernlab)# Create training data (x1, x2: predictors, x3: target)
x1 <- sample(0:1,1000,replace = TRUE)
x2 <- sample(0:1,1000,replace = TRUE)
x3 <- as.numeric(xor(x1,x2))
foo <- runif(1000,min = -0.2,max = 0.2)
x1 <- x1 + foo
foo <- runif(1000,min = -0.2,max = 0.2)
x2 <- x2 + foo# Visualize training data.
plot(cbind(x1,x2),type = "n")
text(cbind(x1,x2),labels = x3)
# Learn SVM and check training error.
foo <- ksvm(cbind(x1,x2),x3,kernel = "vanilladot",type = 'C-svc')
foo# Visualize predictions for training data.
prex3 <- predict(foo,cbind(x1,x2))
plot(cbind(x1,x2),type = "n")
text(cbind(x1,x2),labels = prex3)
```

KERNEL MODELS:Kernel models can also be used for density estimation,to model a probability distribution ordensity function p(x *). In particular, p(x*)=1/n SUMATION k (x* , xi,h) where kernel function k() must integrate 1. To ensure this, you will hereinafter consider k() to be density function of Gaussian distribution

with mean equal to training point xi and standard deviation equal to kernel width h, which evaluated at target point x. *You can get it by using the command dnorm in R. Run code below produce some training data consisting 1500 samples from class 1 and 1000 samples from class 2.These points are stored variables data_class1 and data_class2. Sample 1500 new points from class 1 and 1000 new points from class 2.This validation data. Implement kernel model presented above estimate the density function ofdata sampled from class 1.kernel model should use only training data.Using your implementation of thekernel model, implement a function that computes log likelihood of some data sampled from class 1 function of h.Use this function select h value that maximizes log likelihood of training data from class 1. Use it again select h value that maximizes log likelihood of validation data from class 1.Do these two values coincide? Why or why not?answerfollowing question:Once you have kernel models for p(x* |class=1) and p(x* |class=2),how would you use them to produce posterior class probabilities,p(class|x*)?

```
set.seed(123456789)
N_class1 <- 1500
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)
}
val_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)
  val_class1 <- c(val_class1,b)
}
val_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)
  val_class2 <- c(val_class2,b)
}# p(x*) = (1/n) * sum_{i=1 to n} k(x*, xi, h)
kernel_density <- function(x_star, data, h) {
  density_vals <- sapply(x_star, function(x) {
    mean(dnorm(x, mean = data, sd = h))
  })
  return(density_vals)
}
log_likelihood <- function(data_eval, data_train, h) {
  est_densities <- kernel_density(data_eval, data_train, h)# avoid log(0) by adding small epsilon
  ll <- sum(log(est_densities + 1e-10))
  return(ll)
}
h_vals <- seq(0.1, 5, by = 0.1)
loglik_train <- sapply(h_vals, function(h) log_likelihood(data_class1, data_class1, h))
loglik_val <- sapply(h_vals, function(h) log_likelihood(val_class1, data_class1, h))
best_h_train <- h_vals[which.max(loglik_train)]# Best h values
```

```r
best_h_val <- h_vals[which.max(loglik_val)]
cat("Best h on training data:", best_h_train, "\n")
cat("Best h on validation data:", best_h_val, "\n")
plot(h_vals, loglik_train, type = "l", col = "blue", lwd = 2,
     ylab = "Log-Likelihood", xlab = "h", main = "Log-Likelihood vs h")
lines(h_vals, loglik_val, col = "red", lwd = 2)
legend("bottomright", legend=c("Training", "Validation"),
       col=c("blue", "red"), lwd=2)
# Q: Do the two values of h (one from training, one from validation) coincide? Why or why not?
# A: They may not coincide. The h value maximizing the training log-likelihood
# often leads to overfitting - it models noise and gives high density to
# training points. On the other hand, the h value maximizing validation
# likelihood generalizes better. This is because validation data reflects
# how well the model performs on unseen data. Hence, the training-optimal h
# is often smaller (more peaky), while the validation-optimal h is smoother.
# Posterior class probabilities using kernel models
# Given kernel density models p(x* | class=1) and p(x* | class=2), and class priors,
# we compute: p(class = 1 | x*) = [ p(x* | class=1) * p(class=1) ] /
#                          [ p(x* | class=1)*p(class=1) + p(x* | class=2)*p(class=2) ]
# This gives us the posterior probability that a new point x* belongs to class 1.
# We can similarly compute p(class = 2 | x*) = 1 - p(class = 1 | x*).
```

KERNAL METHODS:Implement such a classifier by using kernel density estimatorand bayes theorm implement a classifier and generalized error,use dataset dataKernel.txt.use gaussian kernel as implemented by R function dnorm,the standard deviation in the function plays role of kernel width h.you want to estimate the generalized error while optimizing the hyperparameter h use 2x2 nested cv :1. Divide the learning data into approximately equal sized folds D1 and D2->2.Divide the fold D1 into approximately equal sized folds D11 and D12->3.For each hyperparameter value h = 0.5, 1, 5, 10 do->4.Train the classifier on D11 and validate it on D12->5.Train the classifier on D12 and validate it on D11->6.Compute the average error on the validation folds->7.Select the hyperparameter value with the lowest average validation error->8.Train the classifier on D1 and test it on D2->9.Repeat the steps 2-8 above swapping the roles of D1 and D2->10.Report the average error on the test folds

```r
# Load required libraries
library(dplyr)# Load the data
data <- read.table("datakernel.txt", header = FALSE)
x <- data[, 1]  # Feature values
y <- data[, 2]  # Class labels (must be numeric or factor)
classes <- unique(y)  # Get unique class labels
# Gaussian kernel function using dnorm
gaussian_kernel <- function(x, x_i, h) {
  dnorm((x - x_i)/h)
}# Kernel density estimator
kde <- function(x, data, h) {
  n <- length(data)
  if (n == 0) return(0)  # Handle empty class case
  (1/n) * sum(gaussian_kernel(x, data, h))
}# Probabilistic classifier using KDE and Bayes theorem
kde_classifier <- function(x_train, y_train, x_test, h) {# Get unique classes
  classes <- unique(y_train)
  n_classes <- length(classes)# Calculate prior probabilities
  priors <- table(y_train)/length(y_train)# Initialize predictions
  preds <- numeric(length(x_test))
```

```
  for (i in seq_along(x_test)) {# Calculate likelihood for each class
    likelihoods <- numeric(n_classes)
    for (c in seq_len(n_classes)) {
      class_data <- x_train[y_train == classes[c]]
      likelihoods[c] <- kde(x_test[i], class_data, h)
    }
    # Calculate posterior probabilities (unnormalized)
    posteriors <- likelihoods * as.numeric(priors)
    # Assign class with highest posterior probability
    preds[i] <- classes[which.max(posteriors)]
  }
  return(preds)
}# Nested 2x2 cross-validation function
nested_cv <- function(x, y, h_values = c(0.5, 1, 5, 10)) {
  n <- length(x)
  if (n != length(y)) stop("x and y must have same length")# First split: D1 and D2 (50/50)
  indices <- sample(n)
  split1 <- floor(n/2)
  D1_idx <- indices[1:split1]
  D2_idx <- indices[(split1+1):n]# Initialize storage
  errors <- numeric(2)
  best_h_values <- numeric(2)# Process both outer folds
  for (outer_fold in 1:2) {
    if (outer_fold == 1) {
      train_idx <- D1_idx
      test_idx <- D2_idx
    } else {
      train_idx <- D2_idx
      test_idx <- D1_idx
    }    # Split train fold into D11 and D12 (50/50 of train)
    n_train <- length(train_idx)
    indices_train <- sample(train_idx)
    split2 <- floor(n_train/2)
    D11_idx <- indices_train[1:split2]
    D12_idx <- indices_train[(split2+1):n_train]# Inner CV to select best h
    best_h <- NA
    best_error <- Inf
    for (h in h_values) {# Train on D11, validate on D12
      preds1 <- kde_classifier(x[D11_idx], y[D11_idx], x[D12_idx], h)
      error1 <- mean(preds1 != y[D12_idx])
      # Train on D12, validate on D11
      preds2 <- kde_classifier(x[D12_idx], y[D12_idx], x[D11_idx], h)
      error2 <- mean(preds2 != y[D11_idx])
      avg_error <- (error1 + error2)/2
      if (avg_error < best_error) {
        best_error <- avg_error
        best_h <- h
      }
    }
    best_h_values[outer_fold] <- best_h# Train on entire train fold with best h and test on test fold
    final_preds <- kde_classifier(x[train_idx], y[train_idx], x[test_idx], best_h)
    errors[outer_fold] <- mean(final_preds != y[test_idx])
  } # Return results
```

```
  return(list(
    avg_test_error = mean(errors),
    best_h_values = best_h_values,
    final_h = median(best_h_values)  # Median is more robust for few values
  ))
}# Execute the analysis
set.seed(123)  # For reproducibility
result <- nested_cv(x, y)# Output the results
cat("Kernel Density Estimation Classifier Results:\n")
cat("=========================================\n")
cat(sprintf("Average test error: %.4f\n", result$avg_test_error))
cat(sprintf("Selected h values in each outer fold: %.1f, %.1f\n",
            result$best_h_values[1], result$best_h_values[2]))
cat(sprintf("Final selected h: %.1f\n", result$final_h))# Create final classifier function with optimiz
final_classifier <- function(new_data) {
  kde_classifier(x, y, new_data, result$final_h)
}
```

KERNEL METHODS-you can see how to produce a probabilistic classifier by using kernel density estimation and Bayes theorem.You are asked to implement such a classifier. The learning data(2500 1-D points with their corresponding class labels)can be obtained via read.table("dataKernel.txt").You should use the Gaussian kernel as implemented by the R function dnorm,the standard deviation in the function plays the role of kernel width h. Divide the data into training (1500 points),validation(500 points)and test(500 points).Use the training and validation data to select among h=0.5,1,5,10 Use test data to estimate the accuracy of model selected.

```
data <- read.table("dataKernel.txt")# Load data
colnames(data) <- c("x", "label")
# Shuffle and split into training (1500), validation (500), test (500)
set.seed(42)
shuffled <- data[sample(nrow(data)), ]
train <- shuffled[1:1500, ]
valid <- shuffled[1501:2000, ]
test  <- shuffled[2001:2500, ]
# Function to compute class-conditional density using Gaussian kernel
kernel_density <- function(x_new, x_train, h) {
  mean(dnorm((x_new - x_train) / h) / h)
}
# Function to compute posterior probabilities using Bayes rule
predict_class <- function(x, train_data, h) {
  class0 <- train_data[train_data$label == 0, "x"]
  class1 <- train_data[train_data$label == 1, "x"]
  # Prior probabilities
  p0 <- length(class0) / nrow(train_data)
  p1 <- length(class1) / nrow(train_data)
  # Likelihoods (via KDE)
  px_given_0 <- kernel_density(x, class0, h)
  px_given_1 <- kernel_density(x, class1, h)
  # Bayes theorem: posterior proportional  prior × likelihood
  post0 <- p0 * px_given_0
  post1 <- p1 * px_given_1
  return(ifelse(post1 > post0, 1, 0))  # Predicted class
}
```

```r
# Try different kernel widths
h_values <- c(0.5, 1, 5, 10)
accuracies <- c()
for (h in h_values) {
  predictions <- sapply(valid$x, predict_class, train_data = train, h = h)
  accuracy <- mean(predictions == valid$label)
  accuracies <- c(accuracies, accuracy)
  cat("Validation Accuracy for h =", h, ":", accuracy, "\n")
}
# Select best h
best_h <- h_values[which.max(accuracies)]
cat("Best h selected:", best_h, "\n")
# Evaluate on test data
final_predictions <- sapply(test$x, predict_class, train_data = train, h = best_h)
test_accuracy <- mean(final_predictions == test$label)
cat("Test Accuracy with h =", best_h, ":", test_accuracy, "\n")
```

You are asked to implement the backpropagation algorithm for fitting the parameters of a neural network (NN) for regression. The NN has one input unit, 10 hidden units, and one output unit. Use the rectifier linear unit (ReLU) activation function, . Recall that you have an example of the backpropagation algorithm in Bishop's book as well as in the course slides. Use the template in the file templateNN.R. Use only basic R functions. Comment your code. Run your code and report the result. Tip 1: The derivative of ReLU is different if aj>0or aj<0and its undefined aj=0. In the latter case, the convention is to use the same derivative as when aj<0 Tip 2: Do not forget to update the bias terms in the backward propagation step. They can be updated as the rest of the weights if you assume that all of them are associated to a dummy input x0=1.

```r
set.seed(123) # For reproducibility
x <- seq(-5, 5, length.out = 100)
y <- x^2 + rnorm(length(x), sd = 2) # Quadratic relationship with noise
x <- (x - mean(x)) / sd(x)
y <- (y - mean(y)) / sd(y)
input_size <- 1
hidden_size <- 10
output_size <- 1
learning_rate <- 0.01
epochs <- 1000
# Initialize weights and biases
# Weights from input to hidden layer
W1 <- matrix(rnorm(input_size * hidden_size, sd = 0.1), nrow = hidden_size)
b1 <- rep(0, hidden_size) # Bias for hidden layer
# Weights from hidden to output layer
W2 <- matrix(rnorm(hidden_size * output_size, sd = 0.1), nrow = output_size)
b2 <- 0 # Bias for output layer
# ReLU activation function
relu <- function(x) {
  return(pmax(0, x))
}
# Derivative of ReLU
relu_derivative <- function(x) {
  return(ifelse(x > 0, 1, 0))
}
# Training the neural network
for (epoch in 1:epochs) {
```

```
  total_loss <- 0
  # Shuffle data (optional for this simple example)
  indices <- sample(length(x))
  for (i in indices) {
    # Forward pass
    # Input to hidden
    a1 <- W1 %*% x[i] + b1
    h1 <- relu(a1)
    # Hidden to output
    a2 <- W2 %*% h1 + b2
    y_pred <- a2 # Linear activation for output (regression)
    # Calculate loss (squared error)
    loss <- (y_pred - y[i])^2 / 2
    total_loss <- total_loss + loss
    # Backward pass
    # Output layer gradient
    d_loss <- (y_pred - y[i])
    # Hidden to output gradients
    dW2 <- d_loss %*% t(h1)
    db2 <- d_loss
    # Backpropagate to hidden layer
    d_h1 <- t(W2) %*% d_loss
    d_a1 <- d_h1 * relu_derivative(a1)
    # Input to hidden gradients
    dW1 <- d_a1 %*% t(x[i])
    db1 <- d_a1
    # Update weights and biases
    W2 <- W2 - learning_rate * dW2
    b2 <- b2 - learning_rate * db2
    W1 <- W1 - learning_rate * dW1
    b1 <- b1 - learning_rate * db1
  }
    # Print loss every 100 epochs
  if (epoch %% 100 == 0) {
    cat("Epoch:", epoch, "Loss:", total_loss / length(x), "\n")
  }
}
# Make predictions
predict_nn <- function(x) {
  a1 <- W1 %*% x + b1
  h1 <- relu(a1)
  a2 <- W2 %*% h1 + b2
  return(a2)
}# Test the network
test_x <- seq(-5, 5, length.out = 20)
test_x_norm <- (test_x - mean(x)) / sd(x) # Use same normalization as training
predictions <- sapply(test_x_norm, predict_nn)
plot(x, y, main = "Neural Network Regression")
lines(test_x_norm, predictions, col = "red", lwd = 2)
legend("topleft", legend = c("Data", "NN Prediction"),
       col = c("black", "red"), lty = c(NA, 1), pch = c(1, NA))
```

KERNEL METHODS-you can see how to produce a probabilistic classifier by using kernel density estimation

and Bayes theorem. You are asked to implement such a classifier. First, you have to produce the learning data by running the code below, which samples 1500 points from class 1 and 1000 points from class 2. These points are stored in the variables data_class1 and data_class2. Second, you have to use the Gaussian kernel and, thus, you have to select an appropriate kernel width h. Choose a value that you deem appropriate. Choose the value manually and disregard overfitting issues. Explain your choice. Finally, compute the posterior distribution of class 1 for the pointsin the interval [-5, 25], e.g. for the points in seq(-5, 25, 0.1). Visualize this distribution with a plot .

```
# Step 1: Generate learning data
set.seed(123)
N_class1 <- 1500
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)
}# Step 2: Define kernel density estimator (Gaussian kernel)
gaussian_kernel <- function(x, data, h) {
  n <- length(data)
  density_est <- sapply(x, function(xi) {
    sum(dnorm((xi - data)/h)) / (n * h)
  })
  return(density_est)
}# Step 3: Compute prior probabilities
prior1 <- N_class1 / (N_class1 + N_class2)
prior2 <- N_class2 / (N_class1 + N_class2)
# Step 4: Compute posterior probability of class 1 for points in [-5, 25]
x_vals <- seq(-5, 25, 0.1)
h <- 1.5  # Chosen manually: smooth but still shows class separation
# Explanation: Chose h = 1.5 to balance smoothness and detail, avoids too much noise
p_x_given_class1 <- gaussian_kernel(x_vals, data_class1, h)
p_x_given_class2 <- gaussian_kernel(x_vals, data_class2, h)
# Apply Bayes' theorem to get P(class 1 | x)
posterior_class1 <- (p_x_given_class1 * prior1) /
  (p_x_given_class1 * prior1 + p_x_given_class2 * prior2)
# Step 5: Plot the posterior probability
plot(x_vals, posterior_class1, type = "l", col = "blue", lwd = 2,
     ylab = "Posterior P(Class 1 | x)", xlab = "x", main = "Posterior Probability of Class 1")
abline(h = 0.5, col = "red", lty = 2)  # Decision boundary
```

NEURAL NETWORKS –You are asked to perform model selection on the iris dataset with the package neuralnet, i.e. the same package that you used in the lab. The iris dataset is included with R. You can find information about the dataset in the help file for the function neuralnet or by typing "?iris". The dataset has 150 cases. Use 50 cases for training, 50 for validation and 50 for test. Be careful when you create these sets as the cases are sorted by class in the dataset. Use the default parameters when learning the neural networks. Select among the following five models: Using only sepal length to predict the species, or using only sepal width, or using only petal length, or using only petal width, or using only sepal length and sepal

width. Estimate the generalization error of the model selected. Produce the model to return to the user. Finally answer the following question. The iris dataset is a classification problem with three classes and, thus, the neural network has to return three probabilities (one for each class) that sum up to 1. You can see this by plotting the neural network learned. You cannot do this with the sigmoid activation function. What activation function do we typically use when we have more than two classes ?

```
library(neuralnet)
data(iris)
# One-hot encode the target variable
iris$setosa <- ifelse(iris$Species == "setosa", 1, 0)
iris$versicolor <- ifelse(iris$Species == "versicolor", 1, 0)
iris$virginica <- ifelse(iris$Species == "virginica", 1, 0)
# Shuffle the data and split into train, validation, and test sets
set.seed(123)
iris <- iris[sample(nrow(iris)), ]
train <- iris[1:50, ]
valid <- iris[51:100, ]
test  <- iris[101:150, ]
# Train candidate models using different input features
models <- list(
  model1 = tryCatch(neuralnet(setosa + versicolor + virginica ~ Sepal.Length,
                              data = train, hidden = 5, linear.output = FALSE),
                    error = function(e) NULL),
  model2 = tryCatch(neuralnet(setosa + versicolor + virginica ~ Sepal.Width,
                              data = train, hidden = 5, linear.output = FALSE),
                    error = function(e) NULL),
  model3 = tryCatch(neuralnet(setosa + versicolor + virginica ~ Petal.Length,
                              data = train, hidden = 5, linear.output = FALSE),
                    error = function(e) NULL),
  model4 = tryCatch(neuralnet(setosa + versicolor + virginica ~ Petal.Width,
                              data = train, hidden = 5, linear.output = FALSE),
                    error = function(e) NULL),
  model5 = tryCatch(neuralnet(setosa + versicolor + virginica ~ Sepal.Length + Sepal.Width,
                              data = train, hidden = 5, linear.output = FALSE), error =
                      function(e) NULL)
)# Function to compute classification error
predict_and_error <- function(model, data) {
  if (is.null(model)) return(Inf)
  pred <- tryCatch(compute(model, data[, model$covariate.names])$net.result, error =
                     function(e) return(NULL))
  if (is.null(pred)) return(Inf)
  pred_class <- apply(pred, 1, which.max)
  true_class <- apply(data[, c("setosa", "versicolor", "virginica")], 1, which.max)
  mean(pred_class != true_class)
}# Evaluate models on validation data
validation_errors <- sapply(models, predict_and_error, data = valid)
# Select best model
best_model_index <- which.min(validation_errors)
best_model <- models[[best_model_index]]
cat("Best model: model", best_model_index, "\n")
cat("Validation error:", validation_errors[best_model_index], "\n")
# Estimate generalization error on test set
test_error <- predict_and_error(best_model, test)
cat("Test error:", test_error, "\n")
```

```r
# Plot the best model
if (!is.null(best_model)) plot(best_model)
# ANSWER TO THE QUESTION (included in code as comment):
# The iris dataset is a classification problem with 3 classes.
# Neural network must output 3 probabilities that sum to 1.
# This cannot be done with the sigmoid activation function.
# Instead, we typically use the SOFTMAX activation function
# for multi-class classification problems to ensure outputs
# are valid probabilities that sum to 1.
```

SUPPORT VECTOR MACHINES-You are asked to use function ksvm from R package kernlab to estimate generalization error of a support vector machine (SVM) for classification of the spam dataset, which included with the package.Use the radial basis function kernel (also known as Gaussian kernel) with a width of 0.05.The C parameter can take value 0.1 or 1. Since the value of the C parameter is not fixed,you need use nested cross-validation to solve this task.

```r
library(kernlab)# Load required packages
library(caret)
data(spam)# Load the spam dataset from kernlab
set.seed(123)  # for reproducibility
# Define outer cross-validation (e.g., 5-fold to estimate generalization error)
outer_folds <- createFolds(spam$type, k = 5, list = TRUE, returnTrain = FALSE)
# Store outer fold errors
outer_errors <- c()
# Outer loop for generalization error estimation
for (i in 1:length(outer_folds)) {
  test_idx <- outer_folds[[i]]
  test_data <- spam[test_idx, ]
  train_data <- spam[-test_idx, ]
  # Inner loop: cross-validation for model selection (C = 0.1, 1)
  inner_folds <- createFolds(train_data$type, k = 3)
  avg_errors <- c()
  for (C_val in c(0.1, 1)) {
    fold_errors <- c()
    for (j in 1:length(inner_folds)) {
      val_idx <- inner_folds[[j]]
      val_data <- train_data[val_idx, ]
      sub_train_data <- train_data[-val_idx, ]
      # Train SVM with Gaussian kernel (RBF), sigma = 0.05, C = C_val
      model <- ksvm(type ~ ., data = sub_train_data,
                    kernel = "rbfdot",
                    kpar = list(sigma = 0.05),
                    C = C_val)
      # Predict on validation set and calculate error
      preds <- predict(model, val_data[,-58])
      fold_errors[j] <- mean(preds != val_data$type)
    }
    # Average validation error for this C
    avg_errors <- c(avg_errors, mean(fold_errors))
  }
  # Select C with lowest average inner error
  best_C <- c(0.1, 1)[which.min(avg_errors)]
  # Train final model on full inner training data with best_C
```

```
  final_model <- ksvm(type ~ ., data = train_data,
                      kernel = "rbfdot",
                      kpar = list(sigma = 0.05),
                      C = best_C)
  # Predict on outer test data and record error
  final_preds <- predict(final_model, test_data[,-58])
  outer_errors[i] <- mean(final_preds != test_data$type)
}# Final generalization error estimate (mean of outer fold errors)
gen_error <- mean(outer_errors)
cat("Estimated generalization error using nested CV:", round(gen_error, 4), "\n")
# Purpose of C parameter:
# The parameter C controls the trade-off between maximizing the margin
# and minimizing the classification error.
# Smaller C → More regularization (wider margin, allows some errors)
# Larger C → Less regularization (tries to classify all points, may overfit)
```

NEURAL NETWORKS - 3 POINTS In the lab on neural networks (NNs), you learned a NN to mimic the sine function. However, you did not estimate the generalization error of the learned NN. You are now asked to do it. Feel free to choose how to do it but note that you already used all the data provided in the lab for learning the NN

```
library(neuralnet)
set.seed(123)
x <- runif(50, min = 0, max = 10)
y <- sin(x)
data <- data.frame(x = x, y = y)
# 5-fold cross-validation
k <- 5
folds <- cut(seq(1, nrow(data)), breaks = k, labels = FALSE)
mse_values <- c()
for (i in 1:k) {
  # Split data
  test_idx <- which(folds == i)
  test_data <- data[test_idx, , drop = FALSE]
  train_data <- data[-test_idx, , drop = FALSE]
  # Scale x to [0, 1] using train data range
  min_x <- min(train_data$x)
  max_x <- max(train_data$x)
  train_data$x_scaled <- (train_data$x - min_x) / (max_x - min_x)
  test_data$x_scaled <- (test_data$x - min_x) / (max_x - min_x)
  # Train model on scaled input
  nn_model <- neuralnet(y ~ x_scaled, data = train_data, hidden = 10, linear.output = TRUE)
  # Make sure model trained successfully
  if (!is.null(nn_model$weights)) {
    # Predict using scaled test input
    pred <- compute(nn_model, test_data["x_scaled"])$net.result
    mse <- mean((pred - test_data$y)^2)
    mse_values <- c(mse_values, mse)
  } else {
    mse_values <- c(mse_values, NA)
    warning(paste("Model training failed for fold", i))
  }
}# Report results
```

```r
cat("Average MSE across", k, "folds:", mean(mse_values, na.rm = TRUE), "\n")
#To estimate the generalization error of the sine-approximating neural network after
#using all data for training, we apply k-fold cross-validation. The dataset is divided
#into k parts; in each round, the network is trained on k-1 parts and tested on the
#remaining one. The process repeats k times, and the average Mean Squared Error (MSE)
#across all folds gives a reliable estimate of generalization error. This approach
#simulates evaluation on unseen data and avoids the need for a separate test set
```

Run the code below to train a neural network (NN) for summing two numbers from the interval[minus 1,1]. Look at the plot of the learned NN and explain why the weights learned make sense. Hints: Note that the activation function is tanh. The two intercepts are so small that you can disregard them. Note that the weights in the first layer are the inverse of the weight in the second layer,0.13=1/7.75

```r
library(neuralnet)
set.seed(1234567890)# Sample 1000 values for x1 and x2 from the interval [-1,1]
x1 <- runif(1000, -1, 1)
x2 <- runif(1000, -1, 1)
tr <- data.frame(x1, x2, y = x1 + x2)# Target output y is simply the sum of x1 and x2
winit <- runif(9, -1, 1)# Random initial weights (not necessary here but shown as in the example)
# Train the neural network with 1 hidden unit, tanh activation function
nn <- neuralnet(formula = y ~ x1 + x2, data = tr, hidden = c(1), act.fct = "tanh")
plot(nn)# Plot the trained neural network
# ---------------------- EXPLANATION OF WHY WEIGHTS MAKE SENSE ----------------------
#The activation function is tanh.:tanh behavesalmostlinearly round 0: tanh(z)=zwhen z is small.
#The two intercepts (biases) are so small that you can disregard them.
#The weights in the first layer are large (e.g., ~7.75), which scales up the input.
#The weight in the second layer is small (~0.13), which is roughly the inverse of 7.75.
#This means:- First layer: z = 7.75*(x1 + x2)
#           - Hidden:     h = tanh(z) =z (for small input range)
#           - Output:     y = 0.13 * h = x1 + x2
# Final Result: The network learns the identity function y = x1 + x2 using scaled tanh,
#               where scaling up and down cancels each other to approximate addition.
```

NEURAL NETWORKS-You are asked to use the function neuralnet of the R package of the same name to train a neural network (NN) to mimic the trigonometric sine function. You should run the following code to obtain the training and test data. Produce the code to train the NN on the training data tr and test it on the data te.Use a single hidden layer with three units. Initialize the weights at random in the interval [-1,1]. Use the default values for the rest of parameters in the function neuralnet. You may need to use the function compute. Confirm that you get results similar to the following figure. Theblack dots are the training data. The blue dots are the test data. The red dots are the NN predictions for the test data. In the previous figure, it is not surprising the poor performance on the range [3,9] because no training point falls in that interval. However, it seems that the predictions convergeto -2 as the value of Var increases. Why do they converge to that particular value ? To answer this question, you may want to look into the weights of the NN learned.

```r
library(neuralnet)
set.seed(1234567890)
train_x <- runif(50, 0, 3)
train_data <- data.frame(Var = train_x, Sin = sin(train_x))
# Test data in [3, 9]
test_x <- runif(50, 3, 9)
test_data <- data.frame(Var = test_x, Sin = sin(test_x))
# Train neural network
```

```r
nn <- neuralnet(Sin ~ Var,
                data = train_data,
                hidden = 3,
                linear.output = TRUE)
# Predict on test data
pred <- compute(nn, test_data)$net.result
# Plot all data
plot(train_data$Var, train_data$Sin, col = "black", pch = 19, ylim = c(-2, 2),
     main = "Neural Network Sine Approximation",
     xlab = "Var", ylab = "Sin(Var)")
# Add test data in blue
points(test_data$Var, test_data$Sin, col = "blue", pch = 19)
# Add NN predictions in red
points(test_data$Var, pred, col = "red", pch = 19)
# ----- EXPLANATION:The neural network is trained only on the interval [0, 3].
# The test data is in [3, 9], where the network has seen no training data.
# Therefore, the model cannot generalize properly beyond the training range,
# and predictions tend to a constant value. In this case, the NN output
# converges to approximately -2 as Var increases, which reflects a kind of
# constant bias learned from the training data.
# This is a common issue: neural networks cannot extrapolate well outside the
# range of training data. They interpolate well but generalize poorly outside.
```

SUPPORT VECTOR MACHINES-You are asked to use the function ksvm from the R package kernlab to learn a support vector machine (SVM) to classify the spam dataset that is included with the package. You should use the radial basis function kernel (also known as Gaussian) with a width of 0.05.You should select the most appropriate value for the C parameter, i.e. you should perform model selection. For this task, you can use any method that you deem appropriate.In the previous question, you may have obtained an error message "no support vectors found" for C = 0. Can you give a plausible explanation for this error ?Estimate the generalization error of the SVM with the C value selected above. Use any method of your choice.Once a SVM has been fitted, a new point is essentially classified according to the sign of a linear combination of support vectors. You are asked to produce the pseudocode (no implementation is required) for computing this linear combination. Your pseudocode should make use of the functions alphaindex, coef and b. See the help of ksvm for information about these functions.

```r
library(kernlab)
data(spam)
spam_data <- spam[, 1:58]
spam_label <- spam[, 58]
dataset <- data.frame(spam_data, type = spam_label)
set.seed(42)# Use 2/3 of data for training, 1/3 for testing
n <- nrow(dataset)
train_index <- sample(1:n, size = round(2 * n / 3))
train_data <- dataset[train_index, ]
test_data <- dataset[-train_index, ]
# Define radial basis function (RBF) kernel with width = 0.05
rbf_kernel <- rbfdot(sigma = 0.05)
C_values <- c(0.01, 0.1, 1, 10)# Try different values for C to perform model selection
accuracies <- c()
for (C in C_values) {
  model <- ksvm(type ~ ., data = train_data, kernel = rbf_kernel, C = C, cross = 5)
  acc <- 1 - model@cross
  accuracies <- c(accuracies, acc)
```

```
}# Choose best C based on highest cross-validation accuracy
best_C <- C_values[which.max(accuracies)]
cat("Best C selected:", best_C, "\n")# Train final model with best C
final_model <- ksvm(type ~ ., data = train_data, kernel = rbf_kernel, C = best_C)
# Predict on test set
predictions <- predict(final_model, test_data)# Estimate generalization error on test set
error_rate <- sum(predictions != test_data$type) / nrow(test_data)
cat("Estimated generalization error:", error_rate, "\n")
# ---- EXPLANATION: Why does C = 0 give error "no support vectors found"? ----
# When C = 0, the SVM model places no penalty on misclassification.
# This means the model allows all errors and does not attempt to find any separating boundary.
# Hence, no support vectors are selected, and the optimization fails.
# SVM requires positive C to balance margin size and classification error.
# ---- PSEUDOCODE: Classification using support vectors ----
# A new point x is classified based on the sign of this expression:
# f(x) = sum_i (alpha_i * K(x_i, x)) + b
# where: - alpha_i are coefficients from coef(model)
# - x_i are support vectors from alphaindex(model)
# - b is the intercept from b(model)
# - K() is the RBF kernel function
# ----- PSEUDOCODE -----
# support_indices <- alphaindex(model)[[1]]
# alphas <- coef(model)[[1]]
# b <- b(model)
# f_x <- 0
# for i in 1:length(support_indices):
#     x_i <- training_data[support_indices[i], ]
#     f_x <- f_x + alphas[i] * RBF_kernel(x_i, x)  # x is new input
# f_x <- f_x + b
# if f_x > 0: classify as "spam"
# else: classify as "non-spam"
# The actual implementation uses built-in prediction, but this is the underlying computation.
```

SUPPORT VECTOR MACHINES-You are asked to use the function ksvm from the R package kernlab to learn a support vector machine (SVM) for classifying the spam dataset that is included with the package.Consider the radial basis function kernel (also known as Gaussian) with a width of 0.05.For the C parameter,consider values 0.5, 1 and 5.This implies that you have to consider three models.Perform model selection, i.e. select the most promising of the three models (use any method of your choice except cross-validation or nested cross-validation).Estimate the generalization error of the SVM selected above (use any method of your choice except cross-validation or nested cross-validation).Produce the SVM that will be returned to the user, i.e. show the code.What is the purpose of the parameter C?

```
library(kernlab)
data(spam)
spam_data <- spam[, c(1:48, 58)]
set.seed(123)
spam_data <- spam_data[sample(nrow(spam_data)), ]
# Split data: 2/3 for training, 1/3 for testing
n <- nrow(spam_data)
train_index <- 1:floor(2 * n / 3)
test_index <- (floor(2 * n / 3) + 1):n
train_data <- spam_data[train_index, ]
test_data <- spam_data[test_index, ]
```

```r
# Fit SVM models with Gaussian kernel width = 0.05 and C = 0.5, 1, 5
model_C_0.5 <- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
                    kpar = list(sigma = 0.05), C = 0.5)
model_C_1 <- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
                  kpar = list(sigma = 0.05), C = 1)
model_C_5 <- ksvm(type ~ ., data = train_data, kernel = "rbfdot",
                  kpar = list(sigma = 0.05), C = 5)
# Predict on the test set for each model
pred_0.5 <- predict(model_C_0.5, test_data[,-58])
pred_1 <- predict(model_C_1, test_data[,-58])
pred_5 <- predict(model_C_5, test_data[,-58])
# Compute test error rates for each model
error_0.5 <- sum(pred_0.5 != test_data$type) / length(pred_0.5)
error_1 <- sum(pred_1 != test_data$type) / length(pred_1)
error_5 <- sum(pred_5 != test_data$type) / length(pred_5)
# Print error rates
cat("Test Error for C=0.5:", error_0.5, "\n")
cat("Test Error for C=1  :", error_1, "\n")
cat("Test Error for C=5  :", error_5, "\n")
# ---- ANSWERS ----
# Model Selection:We select the model with the lowest test error (among C=0.5, 1, 5).
# This is a simple holdout validation method, not cross-validation.
# Assume C=1 has the lowest error, so we select that.
# Generalization Error Estimation:
# We estimate the generalization error using the test set.
# For the selected model (C=1), estimated test error is printed above.
# Final SVM to be returned to user:
final_model <- model_C_1  # Final chosen model with C=1
# Purpose of parameter C:
# C controls the trade-off between margin size and classification error.
# Smaller C → more regularization → wider margin, allows misclassification.
# Larger C → less regularization → tries to classify training data better (can overfit).
```

NEURAL NETWORKS-Train a neural network (NN) to learn the trigonometric sine function. To do so, sample 50 points uniformly at random in the interval [0, 10]. Apply the sine function to each point. The resulting pairs are the data available to you. Use 25 of the 50 points for training and the rest for validation. The validation set is used for early stop of the gradient descent. Consider threshold values i/1000 with i = 1, . . . , 10. Initialize the weights of the neural network to random values in the interval [minus 1, 1]. Consider two NN architectures: A single hidden layer of 10 units, and two hidden layers with 3 units each. Choose the most appropriate NN architecture and threshold value. Motivate your choice. Feel free to reuse the code of the corresponding lab. Estimate the generalization error of the NN selected above (use any method of your choice). In the light of the results above, would you say that the more layers the better ? Motivate your answer.

```r
library(neuralnet)
set.seed(123)
x <- runif(50, min = 0, max = 10)
y <- sin(x)
data <- data.frame(x = x, y = y)
train_idx <- sample(1:50, 25)
train_data <- data[train_idx, ]
val_data <- data[-train_idx, ]
# Step 3: Define training function (repeat attempts)
```

```r
train_nn <- function(train_data, val_data, hidden_layers, threshold, attempts = 3) {
  best_model <- NULL
  best_error <- Inf

  for (i in 1:attempts) {
    nn <- tryCatch({
      neuralnet(y ~ x,
                data = train_data,
                hidden = hidden_layers,
                threshold = threshold,
                startweights = runif(10 * length(hidden_layers) + length(hidden_layers), -1, 1),
                stepmax = 1e6)  # increased from 1e5 to 1e6
    }, error = function(e) NULL)

    if (!is.null(nn) && !is.null(nn$weights) && length(nn$weights) > 0) {
      pred <- tryCatch({
        compute(nn, val_data$x)$net.result
      }, error = function(e) rep(NA, nrow(val_data)))

      if (!any(is.na(pred))) {
        val_error <- mean((val_data$y - pred)^2)
        if (val_error < best_error) {
          best_error <- val_error
          best_model <- nn
        }
      }
    }
  }

  if (is.null(best_model)) {
    return(list(model = NULL, val_error = Inf))
  } else {
    return(list(model = best_model, val_error = best_error))
  }
}
# Step 4: Try architectures and thresholds
results <- list()
i <- 1
thresholds <- seq(0.01, 0.1, length.out = 10)
for (thresh in thresholds) {
  res_1layer <- train_nn(train_data, val_data, hidden_layers = 10, threshold = thresh)
  res_2layer <- train_nn(train_data, val_data, hidden_layers = c(3, 3), threshold = thresh)

  cat(sprintf("Threshold %.3f -> 1-layer error: %.5f, 2-layer error: %.5f\n",
              thresh, res_1layer$val_error, res_2layer$val_error))

  results[[i]] <- list(
    thresh = thresh,
    err_1layer = res_1layer$val_error,
    err_2layer = res_2layer$val_error,
    model_1layer = res_1layer$model,
    model_2layer = res_2layer$model
  )
```

```
  i <- i + 1
}
# Step 5: Select best model
best_result <- NULL
min_error <- Inf
for (res in results) {
  if (!is.infinite(res$err_1layer) && res$err_1layer < min_error) {
    min_error <- res$err_1layer
    best_result <- list(model = res$model_1layer,
                        arch = "1-layer (10 units)",
                        thresh = res$thresh,
                        val_error = res$err_1layer)
  }
  if (!is.infinite(res$err_2layer) && res$err_2layer < min_error) {
    min_error <- res$err_2layer
    best_result <- list(model = res$model_2layer,
                        arch = "2-layer (3+3 units)",
                        thresh = res$thresh,
                        val_error = res$err_2layer)
  }
}
# Step 6: Print and plot
if (!is.null(best_result) && !is.null(best_result$model)) {
  cat("Best model uses architecture:", best_result$arch, "\n")
  cat("Best threshold value:", best_result$thresh, "\n")
  cat("Validation error of best model:", best_result$val_error, "\n")
  plot(best_result$model, rep = "best")
} else {
  cat("No valid models were trained.\n")
}
#Estimate of generalization error:We use the validation error as a proxy for generalization error .
# Estimated generalization error: printed above.
#Is more layers always better?:
# No. In this case, the best model may come from either architecture.
# Deeper networks can overfit small datasets and converge slower.
# Hence, "more layers" is not always better-it depends on data and task complexity.
# Plot the best model
plot(best_result$model, rep = "best")
```

SUPPORT VECTOR MACHINES-In this assignment, you are asked to use the R package kernlab to learn SVMs for classifying the spam dataset that is included with the package. Consider the radial basis function kernel (also known as Gaussian) with a width of 0.05. For the C parameter, consider values 1, 10 and 100. (2p) Estimate the error for the three values of C. Use cross-validation with 2 folds. Hint: Use the argument cross=2 when calling the function ksvm. Use the function cross() to print out the error estimate. Use set.seed(1234567890). (2p) In the previous question, the error estimate may not be mono- tone with respect to the value of C. Explain why this happens.

```
library(kernlab)
data(spam)
set.seed(1234567890)
# (2p) Estimate error for C = 1, 10, 100 using 2-fold cross-validation
# Use RBF kernel (Gaussian) with width = 0.05
# Model with C = 1
```

```r
model_C1 <- ksvm(type ~ ., data = spam, kernel = "rbfdot",
                 kpar = list(sigma = 0.05), C = 1, cross = 2)
error_C1 <- cross(model_C1)
# Model with C = 10
model_C10 <- ksvm(type ~ ., data = spam, kernel = "rbfdot",
                  kpar = list(sigma = 0.05), C = 10, cross = 2)
error_C10 <- cross(model_C10)
# Model with C = 100
model_C100 <- ksvm(type ~ ., data = spam, kernel = "rbfdot",
                   kpar = list(sigma = 0.05), C = 100, cross = 2)
error_C100 <- cross(model_C100)
# Print error estimates
cat("Cross-validation error estimate (2-fold):\n")
cat("C = 1   → Error:", error_C1, "\n")
cat("C = 10  → Error:", error_C10, "\n")
cat("C = 100 → Error:", error_C100, "\n")
#ANSWER: Explanation:Why is the error not always decreasing or increasing with C?
# The error estimate may not be monotonic with respect to C due to:
# Data partitioning in 2-fold CV: with only 2 folds, estimates have high variance.
# Smaller C → more regularization → better generalization, but may underfit.
# Larger C → less regularization → can overfit, reducing training error but worsening generalization.
#Therefore,as C increases,model may first improve then degrade,making error non-monotonic.
```

NEURAL NETWORKS-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. Your task is to estimate the mean squared error of a NN with a single hidden layer of 10 units for the regression task described above. Use cross-validation with 2 folds. For the training, initialize the weights of the NN to random values in the interval [minus 1, 1]. Stop the training when the partial derivatives of the error function are below a threshold value of 0.001.

```r
library(neuralnet)
set.seed(1234567890)
Var <- runif(50, 0, 10)
tr <- data.frame(Var, Sin = sin(Var))
tr1 <- tr[1:25, ]  # Fold 1
tr2 <- tr[26:50, ] # Fold 2
train_mse <- function(train_data, test_data) {
  nn <- neuralnet(Sin ~ Var,
                  data = train_data,
                  hidden = 10,
                  threshold = 0.001,
                  startweights = runif(21, -1, 1),  # 10 hidden units * 2 + 1 bias
                  stepmax = 1e6)
  # Fix: pass test data as data.frame with correct column name
  pred <- compute(nn, data.frame(Var = test_data$Var))$net.result
  mean((test_data$Sin - pred)^2)
}
# Train on Fold 1, test on Fold 2
mse1 <- train_mse(tr1, tr2)
# Train on Fold 2, test on Fold 1
mse2 <- train_mse(tr2, tr1)
# Average the two errors
```

```r
mse_cv <- (mse1 + mse2) / 2
cat("2-fold cross-validation MSE:", mse_cv, "\n")
# The NN has 1 hidden layer with 10 units
# Training stops when gradient < 0.001
# Weights initialized randomly in [-1, 1]
# The estimated generalization error is given by the average MSE
```

ENSEMBLE METHODS 1. Interpret the plot resulting from the code below. (1p)

```r
library(mboost)
bf <- read.csv2("bodyfatregression.csv")
set.seed(1234567890)
m <- blackboost(Bodyfat_percent ~ Waist_cm + Weight_kg, data = bf)
mstop(m)
cvf <- cv(model.weights(m), type = "kfold")
cvm <- cvrisk(m, folds = cvf, grid = 1:100)
plot(cvm)
mstop(cvm)
#INTERPRETATION:The plot shows cross-validated error (y-axis) vs number of boosting iterations (x-axis)
# The error first decreases, then increases, forming a U-shape.
# The optimal stopping point (minimum error) occurs at the number shown by mstop(cvm).
# Stopping here avoids overfitting and gives the best generalization performance.
```

SUPPORT VECTOR MACHINES In the following steps, you are asked to use the R package kernlab to learn a SVM for classifying the spam dataset that is included with the package. For the C parameter, consider values 1 and 5. Consider the radial basis function kernel (also known as Gaussian) and the linear kernel. For the former, consider a width of 0.01 and 0.05. This implies that you have to select among six models.2. Use nested cross-validation to estimate the error of the model selection task described above. Use two folds for inner and outer cross-validation. Note that you only have to implement the outer cross-validation: The inner cross-validation can be performed by using the argument cross=2 when calling the function ksvm. 3. Produce the code to select the model that will be returned to the user.

```r
library(kernlab)
data(spam)
set.seed(1234567890)
spam <- spam[sample(nrow(spam)), ]
n <- nrow(spam)
fold1 <- 1:(n/2)
fold2 <- ((n/2)+1):n
C_values <- c(1, 5)
kernels <- c("rbfdot", "rbfdot", "vanilladot")
sigmas <- c(0.01, 0.05, NA)  # NA for linear kernel
# Function to train using inner 2-fold CV and select best model
train_inner_cv <- function(train_data) {
  errors <- numeric(6)
  models <- list()
  i <- 1
  for (C in C_values) {
    for (k in 1:3) {
      kernel <- kernels[k]
      sigma <- sigmas[k]
      if (kernel == "vanilladot") {
        model <- ksvm(type ~ ., data = train_data, kernel = kernel, C = C, cross = 2)
```

```
      } else {
        model <- ksvm(type ~ ., data = train_data, kernel = kernel,
                      kpar = list(sigma = sigma), C = C, cross = 2)
      }
      errors[i] <- cross(model)
      models[[i]] <- model
      i <- i + 1
    }
  }
  best_index <- which.min(errors)
  return(models[[best_index]])
}# Outer fold 1: train on fold1, test on fold2
model1 <- train_inner_cv(spam[fold1, ])
pred1 <- predict(model1, spam[fold2, ])
error1 <- sum(pred1 != spam$type[fold2]) / length(pred1)
# Outer fold 2: train on fold2, test on fold1
model2 <- train_inner_cv(spam[fold2, ])
pred2 <- predict(model2, spam[fold1, ])
error2 <- sum(pred2 != spam$type[fold1]) / length(pred2)
# Final nested CV error estimate
nested_cv_error <- (error1 + error2) / 2
cat("Nested cross-validation error estimate:", nested_cv_error, "\n")
#EXPLANATION:Inner CV (cross=2) selects the best model among 6 candidates.
# Outer CV (2-fold) estimates generalization error of the selected models.Final result
#is average of the two outer fold errors.
# Use the full data to train the best model selected by inner CV
final_model <- train_inner_cv(spam)
#FINAL ANSWER:The final model is trained on all data using inner CV to select the best kernel and C.
# This model is returned to the user.
final_model
```

NEURAL NETWORKS- Implement the backpropagation algorithm for fitting the parameters of a NN for regression. The NN has one input unit, 10 hidden units, and one output unit. Use the tanh activation function. Feel free to use stochastic or batch gradient descent. Please use only basic R functions in your solution, e.g. sum, tanh. Run your code for 5000 iterations (if time permits) on the training data tr below. A learning rate in the interval [1/25_2,1/25] should work fine. Plot the error on tr as well as on the validation data va, as a function of the number of iterations.

```
# Set seed and generate data
set.seed(1234567890)
Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin = sin(Var))
tr <- trva[1:25, ]  # Training data
va <- trva[26:50, ] # Validation data
# Input and targets
x_train <- matrix(tr$Var, ncol = 1)
y_train <- matrix(tr$Sin, ncol = 1)
x_val <- matrix(va$Var, ncol = 1)
y_val <- matrix(va$Sin, ncol = 1)
# Activation function and its derivative
tanh_act <- function(x) tanh(x)
tanh_deriv <- function(x) 1 - tanh(x)^2
# Network architecture
```

```r
n_input <- 1
n_hidden <- 10
n_output <- 1
n_train <- nrow(x_train)
# Initialize weights randomly in [-1, 1]
W1 <- matrix(runif(n_input * n_hidden, -1, 1), n_input, n_hidden)  # Input to hidden
b1 <- runif(n_hidden, -1, 1)  # Hidden bias
W2 <- matrix(runif(n_hidden * n_output, -1, 1), n_hidden, n_output)  # Hidden to output
b2 <- runif(n_output, -1, 1)  # Output bias
# Learning rate
lr <- 0.01
# To store training and validation errors
train_errors <- numeric(5000)
val_errors <- numeric(5000)
for (iter in 1:5000) {
  # Forward pass (train)
  H_in <- x_train %*% W1 + matrix(b1, n_train, n_hidden, byrow = TRUE)
  H_out <- tanh_act(H_in)
  y_pred <- H_out %*% W2 + matrix(b2, n_train, n_output, byrow = TRUE)
  # Compute training error (MSE)
  train_errors[iter] <- mean((y_pred - y_train)^2)
  # Forward pass (validation)
  H_in_val <- x_val %*% W1 + matrix(b1, nrow(x_val), n_hidden, byrow = TRUE)
  H_out_val <- tanh_act(H_in_val)
  y_val_pred <- H_out_val %*% W2 + matrix(b2, nrow(x_val), n_output, byrow = TRUE)
  # Validation error (MSE)
  val_errors[iter] <- mean((y_val_pred - y_val)^2)
  # Backpropagation
  error_out <- y_pred - y_train  # Output layer error
  dW2 <- t(H_out) %*% error_out
  db2 <- colSums(error_out)
  error_hidden <- (error_out %*% t(W2)) * tanh_deriv(H_in)
  dW1 <- t(x_train) %*% error_hidden
  db1 <- colSums(error_hidden)
  # Update weights and biases
  W2 <- W2 - lr * dW2 / n_train
  b2 <- b2 - lr * db2 / n_train
  W1 <- W1 - lr * dW1 / n_train
  b1 <- b1 - lr * db1 / n_train
}
plot(1:5000, train_errors, type = "l", col = "blue", ylim = range(c(train_errors, val_errors)),
     xlab = "Iterations", ylab = "MSE", main = "Training vs Validation Error")
lines(1:5000, val_errors, col = "red")
legend("topright", legend = c("Train Error", "Validation Error"), col = c("blue", "red"), lty = 1)
# ---- ANSWERS ----
# Implemented backprop manually using only basic R.
# NN has 1 input, 10 hidden tanh units, and 1 output.
# Batch gradient descent is used.
# Code runs for 5000 iterations with learning rate 0.01.
# The plot shows how training and validation errors evolve.
# This gives insight into overfitting, convergence, and performance.
```

ML_HELP

Varun Gurupurandar

2025-06-27

LASSO MODEL

```r
library(glmnet)
library(ggplot2)
# Generic LASSO function
run_lasso <- function(
  csv_file = "your_file_path.csv",      # Path to CSV file
  response_var = "target",              # Name of the target variable
  exclude_vars = NULL,                  # Columns to exclude from predictors
  split_ratio = 0.7,                    # Training data ratio (e.g. 0.7 = 70/30 split)
  scale_data = TRUE,                    # Whether to standardize features
  family_type = "gaussian",             # Model family ("gaussian", "poisson")
  log_lambda_fixed = NULL,              # Optional: Evaluate model at log(lambda)
  use_cv = TRUE                         # If TRUE, use cross-validation. If FALSE, use fixed lambda.
) {
  # Step 1: Load data
  data <- read.csv(csv_file)
  cat("Data loaded with", nrow(data), "rows and", ncol(data), "columns.\n")
  # Step 2: Set up predictors and response
  y <- data[[response_var]]
  X <- data[ , !(names(data) %in% c(response_var, exclude_vars))]
  # Step 3: Scale if needed
  if (scale_data) {
    X <- scale(X)
    cat("Features scaled (z-score standardization).\n")
  }
  X <- as.matrix(X)  # Convert predictors to matrix
  set.seed(123)# Step 4: Train/test split
  n <- nrow(X)
  train_idx <- sample(1:n, size = floor(split_ratio * n))
  X_train <- X[train_idx, ]
  y_train <- y[train_idx]
  X_test  <- X[-train_idx, ]
  y_test  <- y[-train_idx]
  cat("Training size:", length(y_train), " | Test size:", length(y_test), "\n")
  # Step 5: LASSO with or without CV
  if (use_cv) {
    cat("Fitting LASSO model with cross-validation...\n")
    cv_model <- cv.glmnet(
      X_train, y_train,
      alpha = 1,
```

```
    family = family_type,
    standardize = FALSE
  )
  best_lambda <- cv_model$lambda.min
  model <- glmnet(
    X_train, y_train,
    alpha = 1,
    lambda = best_lambda,
    family = family_type,
    standardize = FALSE
  )
  # Plot CV error
  plot(cv_model)
  title(paste("Cross-Validation Error vs Lambda (", family_type, ")", sep=""), line = 2.5)
} else {
  # No CV: use fixed or default lambda
  cat("Fitting LASSO model WITHOUT cross-validation...\n")
  # Set lambda manually or use default
  if (!is.null(log_lambda_fixed)) {
    lambda_fixed <- exp(log_lambda_fixed)
  } else {
    lambda_fixed <- 0.01  # default fixed lambda
  }
  best_lambda <- lambda_fixed
  model <- glmnet(
    X_train, y_train,
    alpha = 1,
    lambda = lambda_fixed,
    family = family_type,
    standardize = FALSE
  )
  cat("Model fitted with fixed lambda =", best_lambda, "\n")
}  # Step 6: Predict and calculate MSE
y_pred_train <- predict(model, X_train)
y_pred_test  <- predict(model, X_test)
mse_train <- mean((y_train - y_pred_train)^2)
mse_test  <- mean((y_test - y_pred_test)^2)
# Step 7: Optionally evaluate at log(lambda_fixed)
if (!is.null(log_lambda_fixed) && use_cv) {
  lambda_val <- exp(log_lambda_fixed)
  model_fixed <- glmnet(
    X_train, y_train,
    alpha = 1,
    lambda = lambda_val,
    family = family_type,
    standardize = FALSE)
  cat("\nCoefficients at log(lambda) =", log_lambda_fixed, ":\n")
  print(coef(model_fixed))
} # Step 8: Extract non-zero coefficients
coef_vector <- coef(model)[-1]
selected_features <- coef_vector[coef_vector != 0]
# Step 9: Output results
cat("\n=========== LASSO Summary ===========\n")
```

```
  cat("Response variable:        ", response_var, "\n")
  cat("Model family:             ", family_type, "\n")
  cat("Used cross-validation:    ", use_cv, "\n")
  cat("Lambda used:              ", best_lambda, "\n")
  cat("Number of non-zero features:", length(selected_features), "\n")
  cat("Training MSE:             ", round(mse_train, 4), "\n")
  cat("Test MSE:                 ", round(mse_test, 4), "\n")
  cat("Selected features and coefficients:\n")
  print(selected_features)
  cat("=====================================\n")
}
```

LOGISTIC MODEL

```
library(caret)
library(ggplot2)
# -------------------- User Settings --------------------
# <<< EDIT HERE: Set the full path to your CSV file
file_path <- "your/path/to/data.csv"
data <- read.csv(file_path)
# <<< EDIT HERE: Specify the target column (binary classification)
target_col <- "Diagnosis"
# <<< EDIT HERE: Select predictor columns (set to NULL for all except target)
predictors <- NULL  # Example: c("CW", "BD") or NULL for auto
# <<< EDIT HERE: Should predictors be scaled before modeling?
do_scale <- TRUE
# <<< EDIT HERE: Set proportion of data to use for training (e.g., 0.7 for 70/30)
train_ratio <- 0.5
# <<< OPTIONAL: Force some coefficients to zero
zero_coef_names <- c()  # Example: c("CW", "BD")
# <<< OPTIONAL: Define a custom loss matrix for computing precision
# loss_matrix <- matrix(c(0, 1,
#                         1, 0), nrow = 2, byrow = TRUE)
# <<< EDIT HERE: Set which class is considered "positive" (used for F1 score)
positive_class <- "M"
# <<< EDIT HERE: Set to TRUE if you want to plot a decision boundary (requires 2 predictors)
plot_boundary <- FALSE
# -------------------- Data Preprocessing --------------------
# Extract target variable
y <- as.factor(data[[target_col]])
# Get predictors based on user setting
if (is.null(predictors)) {
  X <- data[, !(names(data) %in% target_col)]
} else {
  X <- data[, predictors, drop = FALSE]
}
# <<< AUTOMATIC: Remove non-numeric predictors before scaling
non_numeric <- names(X)[!sapply(X, is.numeric)]
if (length(non_numeric) > 0) {
  cat("Dropped non-numeric predictors:", paste(non_numeric, collapse = ", "), "\n")
}
X <- X[, sapply(X, is.numeric)]
```

```r
# Scale predictors if enabled
if (do_scale) {
  X <- as.data.frame(scale(X))
}
# Combine predictors and target
full_data <- cbind(X, Target = y)
# Split into training and testing sets
set.seed(123)
n <- nrow(full_data)
train_indices <- sample(seq_len(n), size = floor(train_ratio * n))
train_data <- full_data[train_indices, ]
test_data <- full_data[-train_indices, ]
# -------------------- Fit Logistic Regression Model --------------------
model <- glm(Target ~ ., data = train_data, family = binomial)
# <<< AUTOMATIC: Show model summary (coefficients, z-scores, etc.)
cat("\n-------------------- Model Summary --------------------\n")
print(summary(model))
# <<< AUTOMATIC: Zero out specific coefficients if requested
coefs <- coef(model)
if (length(zero_coef_names) > 0) {
  for (name in zero_coef_names) {
    if (name %in% names(coefs)) {
      coefs[name] <- 0
    }
  }
}
# -------------------- Prediction Function --------------------
predict_with_coefs <- function(model, newdata, custom_coefs) {
  X_mat <- model.matrix(formula(model), newdata)
  probs <- 1 / (1 + exp(-X_mat %*% custom_coefs))
  return(as.vector(probs))
}
# Predict probabilities on train/test sets
train_probs <- predict_with_coefs(model, train_data, coefs)
test_probs <- predict_with_coefs(model, test_data, coefs)
# Threshold at 0.5 to convert probabilities to binary labels
train_preds <- ifelse(train_probs > 0.5, 1, 0)
test_preds <- ifelse(test_probs > 0.5, 1, 0)
# Convert true class labels to 0/1 for evaluation
train_actual <- as.numeric(train_data$Target == positive_class)
test_actual <- as.numeric(test_data$Target == positive_class)
# -------------------- Evaluation --------------------
# Misclassification error
train_error <- mean(train_preds != train_actual)
test_error <- mean(test_preds != test_actual)
cat("\nTrain Misclassification Error:", round(train_error, 4), "\n")
cat("Test  Misclassification Error:", round(test_error, 4), "\n\n")
# Confusion matrix
conf_mat <- table(Predicted = test_preds, Actual = test_actual)
cat("Confusion Matrix (Test):\n")
print(conf_mat)
# Optional: Use loss matrix to compute weighted precision
compute_precision_loss <- function(actual, predicted, L) {
```

```
  total <- 0
  for (i in seq_along(actual)) {
    total <- total + L[actual[i] + 1, predicted[i] + 1]
  }
  return(total / length(actual))
}
if (exists("loss_matrix") && is.matrix(loss_matrix) &&
    all(dim(loss_matrix) == c(2, 2))) {
  precision_loss <- compute_precision_loss(test_actual, test_preds, loss_matrix)
  cat("\nPrecision (with loss matrix):", round(precision_loss, 4), "\n")
} else {
  cat("\n[Info] No valid loss matrix provided. Skipping loss-based precision.\n")
}# Accuracy
accuracy <- mean(test_preds == test_actual)
cat("Accuracy (Test):", round(accuracy, 4), "\n")
# Manual F1-score function
f1_score <- function(actual, predicted) {
  TP <- sum(actual == 1 & predicted == 1)
  FP <- sum(actual == 0 & predicted == 1)
  FN <- sum(actual == 1 & predicted == 0)
  precision <- ifelse((TP + FP) == 0, 0, TP / (TP + FP))
  recall <- ifelse((TP + FN) == 0, 0, TP / (TP + FN))
  if ((precision + recall) == 0) return(0)
  return(2 * precision * recall / (precision + recall))
}
f1 <- f1_score(test_actual, test_preds)
cat("F1 Score (Test, positive =", positive_class, "):", round(f1, 4), "\n")
# -------------------- Prediction for First Observation --------------------
first_obs_probs <- predict_with_coefs(model, full_data[1, , drop = FALSE], coefs)
cat("\nPredicted probability for 1st observation:", round(first_obs_probs, 4), "\n")
# -------------------- Decision Boundary Plot (Only if 2 Predictors) --------------------
if (plot_boundary && ncol(X) == 2) {
  x1 <- names(X)[1]
  x2 <- names(X)[2]
  beta0 <- coefs[1]
  beta1 <- coefs[2]
  beta2 <- coefs[3]

  boundary_fn <- function(x) {
    return(-(beta0 + beta1 * x) / beta2)
  }

  ggplot(full_data, aes_string(x = x1, y = x2, color = target_col)) +
    geom_point() +
    stat_function(fun = boundary_fn, color = "black", linetype = "dashed") +
    ggtitle("Logistic Regression Decision Boundary")
}
```

RIDGE MODEL

```
library(glmnet)  # For ridge classifier (logistic)
library(caret)   # For splitting with stratification
```

```r
# 2. === USER SETTINGS ===
# ------------------------------
# Path to your data file (CSV)
data_path <- "your_data.csv"

# Name of the binary target column
target_col <- "Death"
# Proportion of data to use for training (0 < train_split < 1)
train_split <- 0.5
# Should the predictors be scaled? TRUE or FALSE
scale_features <- TRUE
# Use cross-validation to select best lambda? TRUE or FALSE
use_cv <- TRUE
# If NOT using CV, set lambda manually here
manual_lambda <- 1
# 3. === LOAD YOUR DATA ===
# ------------------------------
# Read your data
data <- read.csv(data_path)
# Separate predictors (X) and target (y)
X <- data[, !(names(data) %in% c(target_col))]  # All columns except target
y <- as.factor(data[[target_col]])              # Make sure target is a factor
# 4. === OPTIONAL: SCALE FEATURES ===
# ------------------------------
# If TRUE, center and scale all predictors to mean 0, sd 1
if (scale_features) {
  X <- scale(X)
}

# Convert to matrix (required by glmnet)
X <- as.matrix(X)
# 5. === SPLIT INTO TRAIN & TEST ===
# ------------------------------
# For reproducibility
set.seed(42)
# Stratified split using caret's createDataPartition
train_idx <- createDataPartition(y, p = train_split, list = FALSE)
# Training data
X_train <- X[train_idx, ]
y_train <- y[train_idx]
# Test data
X_test  <- X[-train_idx, ]
y_test  <- y[-train_idx]
# 6. === FIT RIDGE CLASSIFIER ===
# ------------------------------
# alpha = 0 means Ridge penalty (L2)
# family = "binomial" => logistic regression for binary target
if (use_cv) {
  # If using cross-validation, use cv.glmnet
  cat("Fitting Ridge classifier with CV to find optimal lambda...\n")
  ridge_cv <- cv.glmnet(
    X_train, y_train,
    alpha = 0,            # Ridge penalty
```

```
    family = "binomial",  # Logistic regression
    type.measure = "class" # Use misclassification error for CV
  )  # Get best lambda from CV
  best_lambda <- ridge_cv$lambda.min
  cat("Optimal lambda from CV:", best_lambda, "\n")

} else {
  # If NOT using CV, use glmnet with manual lambda
  cat("Fitting Ridge classifier with manual lambda...\n")

  ridge_cv <- glmnet(
    X_train, y_train,
    alpha = 0,            # Ridge penalty
    family = "binomial",  # Logistic regression
    lambda = manual_lambda
  )
  best_lambda <- manual_lambda
  cat("Manual lambda used:", best_lambda, "\n")
}
# 7. === PREDICT ON TEST SET ===
# ------------------------------
# Predict class probabilities
pred_probs <- predict(ridge_cv, newx = X_test, s = best_lambda,
                      type = "response")
# Convert probabilities to binary class predictions (threshold = 0.5)
y_pred <- ifelse(pred_probs > 0.5, 1, 0)
# Convert actual test labels to numeric (0/1)
y_test_num <- as.numeric(as.character(y_test))
# Create confusion matrix: rows = Predicted, cols = Actual
conf_matrix <- table(Predicted = y_pred, Actual = y_test_num)
cat("\n=== Confusion Matrix ===\n")
print(conf_matrix)
# 8. === APPLY CUSTOM COST MATRIX ===
# ------------------------------
# For binary classification:
# Format:      Actual
#         |   0   |   1
# --------|-------|------
# Pred 0  |  TN   |  FN
# Pred 1  |  FP   |  TP
# Extract confusion matrix elements
TN <- ifelse(!is.na(conf_matrix["0","0"]), conf_matrix["0","0"], 0)
FP <- ifelse(!is.na(conf_matrix["1","0"]), conf_matrix["1","0"], 0)
FN <- ifelse(!is.na(conf_matrix["0","1"]), conf_matrix["0","1"], 0)
TP <- ifelse(!is.na(conf_matrix["1","1"]), conf_matrix["1","1"], 0)
# Define cost matrix:
# [ [0, 1],
#   [10, 0] ]
# So:Cost for FP = 1 per case
# Cost for FN = 10 per case
# TN and TP cost = 0
total_cost <- (FP * 1) + (FN * 10)
cat("\n=== Total cost with cost matrix ===\n")
```

```r
cat("FP:", FP, " x 1 + FN:", FN, " x 10 => Total cost =", total_cost, "\n")
# Optional: Show cost-adjusted matrix for illustration
adjusted_matrix <- matrix(
  c(
    TN * 0, FP * 1,
    FN * 10, TP * 0
  ),
  nrow = 2,
  byrow = TRUE,
  dimnames = list(Predicted = c("0", "1"), Actual = c("0", "1"))
)
cat("\n=== Cost-adjusted Confusion Matrix ===\n")
print(adjusted_matrix)
```

KNN MODEL

```r
# Load libraries
library(caret)
library(kknn)
library(class)
library(ggplot2)
# USER SETTINGS - EDIT THESE
data_path <- "YOUR_DATA.csv"        # <-- Set your data file path
target <- "TARGET_VARIABLE"         # <-- Set your target column name
feature_pattern <- NULL             # Optional: "Channel", etc - or NULL for all except target
use_pca <- TRUE                     # TRUE or FALSE
num_pcs <- 2                        # Number of principal components if PCA is used
train_ratio <- 0.4                  # Training proportion
val_ratio   <- 0.3                  # Validation proportion
test_ratio  <- 0.3                  # Test proportion
k_values <- 1:100                   # K values to test
train_sizes <- NULL                 # E.g., seq(100, 2500, 100) or NULL
knn_mode <- "regress"               # "regress" for regression, "classify" for classification
# Load data
df <- read.csv(data_path)
# Pick features
if (!is.null(feature_pattern)) {
  features <- grep(feature_pattern, names(df), value=TRUE)
} else {
  features <- names(df)[!names(df) %in% target]
}
# PCA if needed
if (use_pca) {
  pc <- prcomp(df[ , features], scale. = TRUE)
  X <- as.data.frame(pc$x[, 1:num_pcs])
} else {
  X <- df[ , features]
}

y <- df[[target]]
# Train/val/test split
set.seed(42)
```

```r
train_idx <- createDataPartition(y, p=train_ratio, list=FALSE)
remaining_idx <- setdiff(seq_len(nrow(df)), train_idx)
val_idx <- sample(remaining_idx, size=val_ratio * nrow(df))
test_idx <- setdiff(remaining_idx, val_idx)
X_train <- X[train_idx, , drop=FALSE]
y_train <- y[train_idx]
X_val   <- X[val_idx, , drop=FALSE]
y_val   <- y[val_idx]
X_test  <- X[test_idx, , drop=FALSE]
y_test  <- y[test_idx]
# Scale
# ================================================
X_train_scaled <- scale(X_train)
X_val_scaled   <- scale(X_val,  center=attr(X_train_scaled, "scaled:center"), scale=attr(X_train_scaled
X_test_scaled  <- scale(X_test, center=attr(X_train_scaled, "scaled:center"), scale=attr(X_train_scaled
# Generic KNN function using kknn
# ================================================
run_knn <- function(k) {
  if (knn_mode == "regress") {
    df_train <- data.frame(X_train_scaled, y = y_train)
    df_val   <- data.frame(X_val_scaled)
    df_test  <- data.frame(X_test_scaled)

    model_train <- kknn(y ~ ., train = df_train, test = df_train, k = k, kernel = "rectangular")
    model_val   <- kknn(y ~ ., train = df_train, test = df_val,   k = k, kernel = "rectangular")
    model_test  <- kknn(y ~ ., train = df_train, test = df_test,  k = k, kernel = "rectangular")

    pred_train <- fitted(model_train)
    pred_val   <- fitted(model_val)
    pred_test  <- fitted(model_test)

    c(
      k = k,
      Train = mean((y_train - pred_train)^2),
      Val   = mean((y_val   - pred_val)^2),
      Test  = mean((y_test  - pred_test)^2)
    )
  } else {
    df_train <- data.frame(X_train_scaled, y = y_train)
    df_val   <- data.frame(X_val_scaled)
    df_test  <- data.frame(X_test_scaled)

    model_train <- kknn(y ~ ., train = df_train, test = df_train, k = k, kernel = "rectangular")
    model_val   <- kknn(y ~ ., train = df_train, test = df_val,   k = k, kernel = "rectangular")
    model_test  <- kknn(y ~ ., train = df_train, test = df_test,  k = k, kernel = "rectangular")

    pred_train <- fitted(model_train)
    pred_val   <- fitted(model_val)
    pred_test  <- fitted(model_test)

    c(
      k = k,
      Train = mean(pred_train == y_train),
```

```
      Val   = mean(pred_val   == y_val),
      Test  = mean(pred_test  == y_test)
    )
  }
}
# Run for all K
# ================================================
results <- t(sapply(k_values, run_knn))
print(results)
# Learning curve if varying training size
# ================================================
if (!is.null(train_sizes)) {
  metric_train <- c()
  metric_test  <- c()

  for (n in train_sizes) {
    idx <- seq_len(min(n, length(train_idx)))
    X_sub <- X_train[idx, , drop=FALSE]
    y_sub <- y_train[idx]
    X_sub_scaled <- scale(X_sub)
    X_test_scaled2 <- scale(X_test, center=attr(X_sub_scaled,"scaled:center"), scale=attr(X_sub_scaled,

    df_sub <- data.frame(X_sub_scaled, y = y_sub)
    df_test2 <- data.frame(X_test_scaled2)

    if (knn_mode == "regress") {
      model_train <- kknn(y ~ ., train = df_sub, test = df_sub, k = 10, kernel = "rectangular")
      model_test  <- kknn(y ~ ., train = df_sub, test = df_test2, k = 10, kernel = "rectangular")

      pred_train <- fitted(model_train)
      pred_test  <- fitted(model_test)

      metric_train <- c(metric_train, mean((y_sub - pred_train)^2))
      metric_test  <- c(metric_test, mean((y_test - pred_test)^2))
    } else {
      model_train <- kknn(y ~ ., train = df_sub, test = df_sub, k = 10, kernel = "rectangular")
      model_test  <- kknn(y ~ ., train = df_sub, test = df_test2, k = 10, kernel = "rectangular")

      pred_train <- fitted(model_train)
      pred_test  <- fitted(model_test)

      metric_train <- c(metric_train, mean(pred_train == y_sub))
      metric_test  <- c(metric_test, mean(pred_test == y_test))
    }
  }

  df_sizes <- data.frame(Size = train_sizes, Train = metric_train, Test = metric_test)
  ggplot(df_sizes, aes(x = Size)) +
    geom_line(aes(y = Train, color = "Train")) +
    geom_line(aes(y = Test, color = "Test")) +
    labs(title = "Performance vs Training Size") +
    theme_minimal()
}
```

```r
# Plot metric vs K
# ================================================
df_k <- as.data.frame(results)
metric <- ifelse(knn_mode == "regress", "MSE", "Accuracy")

ggplot(df_k, aes(x = as.numeric(k))) +
  geom_line(aes(y = Train, color = "Train")) +
  geom_line(aes(y = Val, color = "Validation")) +
  geom_line(aes(y = Test, color = "Test")) +
  labs(title = paste0(metric, " vs K"), x = "K", y = metric) +
  theme_minimal()
```