STATISTICAL LEARNING

Prabhjot Prabhjot

(B34000105)

IMPLEMENTING AND TESTING K-NEAREST NEIGHBORS

What are K-Nearest Neighbors?

The k-Nearest Neighbors (k-NN) algorithm is a simple, yet powerful, non-parametric supervised learning method used for classification . The k-NN algorithm classifies a data point based on the majority class among its k-nearest neighbors in the feature space. It is based on the idea that similar data points are likely to be close to each other in the feature space.

Steps for performing K-NN classification

1. **Select the Number of Neighbors (k)**:

- The first step in the k-NN algorithm is to choose the value of k, which is the number of nearest neighbors to consider when making the classification decision.

2. **Calculate the Distance**:

- For each data point in the dataset, the distance between the query (test point) and all the training data points is calculated. Common distance metrics include Euclidean distance, Manhattan distance, and Hammer distance.

3. **Identify Nearest Neighbors**:

- The k data points from the training set that have the smallest distance to the query point are identified as the nearest neighbors.

4. **Majority Voting (for Classification)**:

- The class labels of the k-nearest neighbors are analyzed. The most frequent class label among these neighbors is assigned to the query point. This is known as majority voting.

5. **Assign Class Label**:

- The class with the highest vote among the k-nearest neighbors is assigned as the predicted class for the query point.

OBJECTIVE OF OUR PROJECT

The objective of this project is to implement, test, and compare a custom k-Nearest Neighbors (k-NN) classifier with the built-in k-NN function provided by the class package in R. We will also modify the custom k-NN classifier to work with different values of k and evaluate its performance on the Iris dataset.

STEPS AND IMPLEMENTATIONS

Part 1: Project Setup

- We began by loading the necessary libraries: datasets for accessing the Iris dataset and class for the built-in knn function.
- The Iris dataset was loaded, and the features (X) and target labels (Y) were assigned.

Part 2: Implementing the Custom PKV Function

- The PKV function was implemented to perform k-Nearest Neighbors classification.
- For each test point, the Euclidean distance to all other points was calculated.
- The K nearest neighbors were identified, and the majority class among these neighbors was chosen as the predicted label

Part 3: Testing on the Iris Dataset

- The custom PKV function was tested on the Iris dataset with K = 1.
- The first few predictions were displayed to verify the function's output.
- The error rate for the custom 1-NN was calculated and printed. The result was an error rate of 4% (0.04).

Part 4: Testing the Built-in k-Nearest Neighbors Function

- The built-in Knn function from the class package was used for comparison with K = 1.
- The error rate for the built-in 1-NN was calculated and found to be 0%, indicating perfect classification.
- A comparison was made to check if the results of the custom and built-in functions were different. The result was TRUE, indicating a difference in error rates.

Part 5: Modifying the PKV Function for Different Values of K

- The custom PKV function was tested with different values of K (1, 4, 5).
- The error rates for each value of K were calculated and displayed:
    - For K = 1, the error rate was 4% (0.04).
    - For K = 4, the error rate was also 4% (0.04).
    - For K = 5, the error rate decreased to approximately 3.33% (0.0333).

```r
# Part 1: Project Setup
# Load necessary libraries
library(datasets)
library(class)  # for the built-in knn function

# Load the Iris dataset
data(iris)

# Assign features to X
X <- iris[, 1:4]

# Assign target variable to Y
Y <- iris$Species

# Part 2: Implementing the Custom PKV Function
# Implement the PKV function
PKV <- function(X, Y, K = 1) {
  predictions <- sapply(1:nrow(X), function(i) {
    # Extract the test point
    test_point <- X[i, ]

    # Calculate Euclidean distances manually
    distances <- apply(X[-i, ], 1, function(row) {
      sqrt(sum((row - test_point) ^ 2))
    })

    # Find the K nearest neighbors (the ones with the minimum distances)
    nearest_neighbors <- order(distances)[1:K]

    # Predict the label by majority vote among the K nearest neighbors
    predicted_labels <- Y[-i][nearest_neighbors]
    predicted_label <- names(sort(table(predicted_labels), decreasing = TRUE))[1]

    return(predicted_label)
  })
  return(predictions)
}

# Calculate and return the prediction error
PKV_error <- function(X, Y, K = 1) {
  predictions <- PKV(X, Y, K)

  # Check for NA in predictions
  if (any(is.na(predictions))) {
    print("NA found in predictions")
    print(predictions)
  }

  error_rate <- mean(predictions != Y, na.rm = TRUE)
  return(error_rate)
}

# Part 3: Testing on the Iris Dataset
# Test the PKV function on the Iris dataset
predictions <- PKV(X, Y, 1)

# Display the first few predictions
print(head(predictions))

# Calculate the prediction error for K = 1
error_rate <- PKV_error(X, Y, 1)
print(paste("Custom 1-NN Error Rate on Iris dataset:", error_rate))

# Part 4: Testing the Built-in k-Nearest Neighbors Function
# Compare with built-in knn function with K = 1
knn_builtin_1 <- knn(train = X, test = X, cl = Y, k = 1)
builtin_error_rate_1 <- mean(knn_builtin_1 != Y)
print(paste("Built-in 1-NN Error Rate:", builtin_error_rate_1))

# Test the PKV function with K = 1
error_rate_1 <- PKV_error(X, Y, 1)
print(paste("Custom 1-NN Error Rate:", error_rate_1))

# Check if results are different for K = 1
print("Are the results different for K = 1?")
print(builtin_error_rate_1 != error_rate_1)

# Part 5: Modifying the PKV Function for Different Values of K
# Test the modified PKV function with different values of K
k_values <- c(1, 4, 5)
for (k in k_values) {
  error_rate_k <- PKV_error(X, Y, k)
  print(paste(k, "-NN Error Rate:", error_rate_k))
}
```

```r
# Additional Comparison
# Ensure predictions_custom and predictions_builtin are defined correctly
predictions_custom <- PKV(X, Y, 1)
predictions_builtin <- knn(train = X, test = X, cl = Y, k = 1)

# Compare predictions for all samples
comparison <- predictions_custom != predictions_builtin

print("Indices with different predictions:")
print(which(comparison))

print("Custom predictions for these indices:")
print(predictions_custom[which(comparison)])

print("Built-in predictions for these indices:")
print(predictions_builtin[which(comparison)])
```

RESULTS

```
> print(paste("Custom 1-NN Error Rate on Iris dataset:", error_rate))
[1] "Custom 1-NN Error Rate on Iris dataset: 0.04"

> print(paste("Built-in 1-NN Error Rate:", builtin_error_rate_1))
[1] "Built-in 1-NN Error Rate: 0"

> print("Are the results different for K = 1?")
[1] "Are the results different for K = 1?"
> print(builtin_error_rate_1 != error_rate_1)
[1] TRUE
```

- **Custom Implementation**:
  - The custom k-NN classifier implementation had a higher error rate compared to the built-in function, likely due to potential inefficiencies or inaccuracies in the distance calculations and neighbor voting process.
- **Built-in Implementation**:
  - The built-in k-NN classifier from the class package performed perfectly with an error rate of 0%, highlighting the robustness and reliability of well-tested library functions.

```
+   print(paste(k, "-NN Error Rate:", error_rate_k))
+ }
[1] "1 -NN Error Rate: 0.04"
[1] "4 -NN Error Rate: 0.04"
[1] "5 -NN Error Rate: 0.0333333333333333"
```

- **Effect of K on Error Rate**:
  - Increasing the value of K slightly improved the error rate, reducing it to approximately 3.33% for K = 5.

Additional Comparison

- Predictions from both the custom and built-in functions were compared to identify differences.
- Indices with different predictions were printed.
- Custom and built-in predictions for these differing indices were displayed to understand the discrepancies.

```
> print("Indices with different predictions:")
[1] "Indices with different predictions:"
> print(which(comparison))
[1]  71  73  84 107 120 134
> print("Custom predictions for these indices:")
[1] "Custom predictions for these indices:"
> print(predictions_custom[which(comparison)])
[1] "virginica"  "virginica"  "virginica"  "versicolor" "versicolor" "versicolor"
> print("Built-in predictions for these indices:")
[1] "Built-in predictions for these indices:"
> print(predictions_builtin[which(comparison)])
[1] versicolor versicolor versicolor virginica  virginica  virginica
Levels: setosa versicolor virginica
```

SUMMARY

This project successfully implemented, tested, and compared a custom k-NN classifier with the built-in k-NN function on the Iris dataset. By exploring different values of K, we gained a deeper understanding of the k-NN algorithm's behavior and the impact of neighbor count on classification performance. The built-in function's superior performance underscored the value of using established libraries for machine learning tasks.

IMPLEMENTATION AND EVALUATION OF NAIVE BAYES CLASSIFIER

WHAT IS NAÏVE BAYES CLASSIFIER?

The Naive Bayes classifier is a simple yet powerful probabilistic machine learning algorithm used for classification tasks. It is based on Bayes' Theorem and assumes that the features (attributes) in a dataset are conditionally independent given the class label.

For a given data point $x=(x1,x2,\ldots,xn)$ with features $x1,x2,\ldots,xn$ the Naive Bayes classifier predicts the class ωk that maximizes the posterior probability $P(\omega k|x)$. According to Bayes' Theorem:

$P(\omega k|x)= P(x|\omega k)\cdot P(\omega k)/P(x)$

Since P(x)P(x) is constant for all classes, it can be ignored in the maximization process, resulting in:

class(x)=arg max ωk (P(x|ωk)·P(ωk))

STEPS IN IMPLEMENTING A NAIVE BAYES CLASSIFIER

 Calculate Prior Probabilities: Determine the prior probabilities for each class based on the training data.

 Calculate Likelihoods: For each feature, compute the likelihood (conditional probability) for each class.

- For Gaussian Naive Bayes, this involves calculating the mean and standard deviation for each feature per class.
- For Multinomial or Bernoulli Naive Bayes, this involves calculating the probabilities of each feature occurring in each class.

 Apply Bayes' Theorem: Use the prior probabilities and likelihoods to calculate the posterior probability for each class given a data point.

 Make Predictions: Assign the class with the highest posterior probability to the data point.

OBJECTIVE

The objective of this project is to implement and evaluate the Naive Bayes Classifier in R. We aim to create a custom Naive Bayes Classifier and compare its performance with the built-in naiveBayes function from the e1071 package.

STEPS AND IMPLEMENTATIONS

Step 1: Create the Naive Bayes Classifier Function

 Calculate Class Centroids and Prior Probabilities:

- For each class, calculate the centroid (mean of each feature).
- Calculate the prior probability of each class (proportion of instances in each class).

 Distance-Based Prediction:

- For each data point, calculate the distance to each class centroid.
- Predict the class with the smallest distance weighted by the prior probabilities.

Step 2: Modify the Function to Calculate Prediction Error

Calculate Prediction Error:

- Compare the predicted labels to the actual labels.
- Calculate the error rate as the percentage of incorrect predictions.

Step 3: Test the Built-in Naive Bayes Classifier

1. **Use naiveBayes Function from e1071 Package**:
    - Train the Naive Bayes classifier using the naiveBayes function.
    - Predict the labels for the dataset.
    - Calculate the error rate for the predictions.

Step 4: Modify the Function to Use Gaussian Distribution

Gaussian-Based Prediction:

- Assume that each feature follows a Gaussian (normal) distribution.
- Calculate the mean and standard deviation of each feature for each class.
- Use the Gaussian probability density function to calculate the likelihood of each feature value given the class.
- Predict the class with the highest posterior probability.

```r
# Load necessary library
library(e1071)

# Function to calculate the Naive Bayes Classifier
NBC <- function(X, Y) {
  # Get unique classes
  classes <- unique(Y)
  # Initialize vectors for class probabilities and centroids
  class_probs <- numeric(length(classes))
  centroids <- matrix(0, nrow = length(classes), ncol = ncol(X))

  # Calculate the centroids and prior probabilities for each class
  for (i in 1:length(classes)) {
    class_data <- X[Y == classes[i],]
    centroids[i,] <- colMeans(class_data)
    class_probs[i] <- nrow(class_data) / nrow(X)
  }

  # Function to predict the class for a single data point
  predict_class <- function(x) {
    distances <- apply(centroids, 1, function(centroid) sum((x - centroid)^2))
    probs <- class_probs / distances
    return(classes[which.max(probs)])
  }

  # Predict class for each data point in X
  predictions <- apply(X, 1, predict_class)
  return(predictions)
```

```r
# Example usage with Iris dataset
data(iris)
X <- iris[, -5]
Y <- iris[, 5]
predictions <- NBC(X, Y)
print(predictions)


#step2 # Function to calculate the Naive Bayes Classifier with prediction error
NBC <- function(X, Y) {
  classes <- unique(Y)
  class_probs <- numeric(length(classes))
  centroids <- matrix(0, nrow = length(classes), ncol = ncol(X))

  for (i in 1:length(classes)) {
    class_data <- X[Y == classes[i],]
    centroids[i,] <- colMeans(class_data)
    class_probs[i] <- nrow(class_data) / nrow(X)
  }

  predict_class <- function(x) {
    distances <- apply(centroids, 1, function(centroid) sum((x - centroid)^2))
    probs <- class_probs / distances
    return(classes[which.max(probs)])
  }

  predictions <- apply(X, 1, predict_class)
  error_rate <- mean(predictions != Y)


  return(list(predictions = predictions, error_rate = error_rate))
}

# Example usage with Iris dataset
results <- NBC(X, Y)
print(results$predictions)
print(paste("Error rate:", results$error_rate))

#step3
# Load e1071 library
library(e1071)

# Train the model using the e1071 package
model <- naiveBayes(X, Y)
predictions_e1071 <- predict(model, X)
error_rate_e1071 <- mean(predictions_e1071 != Y)

print(predictions_e1071)
print(paste("Error rate using e1071:", error_rate_e1071))

#step4:
# Function to calculate the Naive Bayes Classifier using Gaussian distribution
NBC_gaussian <- function(X, Y) {
  classes <- unique(Y)
  class_probs <- numeric(length(classes))
  class_means <- matrix(0, nrow = length(classes), ncol = ncol(X))
  class_sds <- matrix(0, nrow = length(classes), ncol = ncol(X))


  for (i in 1:length(classes)) {
    class_data <- X[Y == classes[i],]
    class_means[i,] <- colMeans(class_data)
    class_sds[i,] <- apply(class_data, 2, sd)
    class_probs[i] <- nrow(class_data) / nrow(X)
  }

  gaussian_prob <- function(x, mean, sd) {
    return(exp(-0.5 * ((x - mean) / sd)^2) / (sqrt(2 * pi) * sd))
  }

  predict_class <- function(x) {
    probs <- sapply(1:length(classes), function(i) {
      prod(gaussian_prob(x, class_means[i,], class_sds[i,])) * class_probs[i]
    })
    return(classes[which.max(probs)])
  }

  predictions <- apply(X, 1, predict_class)
  error_rate <- mean(predictions != Y)
  return(list(predictions = predictions, error_rate = error_rate))
}

# Example usage with Iris dataset
results_gaussian <- NBC_gaussian(X, Y)
print(results_gaussian$predictions)
print(paste("Error rate using Gaussian distribution:", results_gaussian$error_rate))
```

RESULTS

```
> print(paste("Error rate using e1071:", error_rate_e1071))
[1] "Error rate using e1071: 0.04"
> print(paste("Error rate using Gaussian distribution:", results_gaussian$error_rate))
[1] "Error rate using Gaussian distribution: 0.04"
```

- **Error Rate using e1071 Package**: 0.04 (4%)
- **Error Rate using Gaussian Distribution**: 0.04 (4%)

Interpretation of Results

- Both the custom Naive Bayes classifier using Gaussian distribution and the built-in naiveBayes function from the e1071 package produced an identical error rate of 4% on the Iris dataset.

- This demonstrates that the custom implementation is correct and performs equivalently to the standard implementation.

- The Gaussian assumption for feature distributions is appropriate for the Iris dataset and yields reliable classification results.

CONCLUSION

The project successfully implemented and evaluated the Naive Bayes Classifier in R. Both the custom Gaussian-based implementation and the e1071 package's implementation performed equally well, with an error rate of 4%. This validates the effectiveness of the Naive Bayes approach and the Gaussian assumption for this dataset. Further testing on different datasets could provide additional insights into the classifier's performance and robustness.