Lecture 13

KNN, Distance metrics

Let's start with an overview of KNN, or K-Nearest Neighbors.

What is KNN?

- K-Nearest Neighbors (KNN) is a simple, non-parametric, and lazy machine learning algorithm primarily used for classification tasks but can also be applied to regression.
- **Non-parametric** means that it does not assume a fixed form (or distribution) for the data, making it versatile for various types of datasets.
- **Lazy algorithm** implies that it makes no assumptions during the training phase and instead waits until a query point needs to be predicted.

How Does KNN Work?

- Given a query point, KNN looks for the 'K' closest data points in the training dataset and uses their labels (in classification) or values (in regression) to make a prediction.
- The core idea: **"Birds of a feather flock together."** That is, similar instances are likely to have similar labels.

While KNN is typically deployed as a classification algorithm, it can be modified to work as a regression algorithm. We'll look at this case in a future lecture, but for the moment, we'll stick with the classification context.

We looked in a prior lecture at distance metrics for this and other related applications. The way that KNN determines closeness or similarity is done by distance metrics. Let's review some common choices used with KNN.

What Are Distance Metrics?

- The concept of "closeness" is defined using distance metrics. Different distance metrics can lead to different results.

Common Distance Metrics:

- *Euclidean Distance*: Most common and intuitive. $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
- *Manhattan Distance*: Sum of absolute differences. $d(x, y) = \sum_{i=1}^{n}|x_i - y_i|$
- *Minkowski Distance*: A generalization of both Euclidean and Manhattan. $d(x, y) = \sqrt[p]{\sum_{i=1}^{n}|x_i - y_i|^p}$
- *Chebyshev Distance*: The maximum difference along any coordinate dimension. $d(x, y) = \max(|x_i - y_i|)$
- *Cosine Similarity*: Measures the cosine of the angle between two vectors (used when the magnitude of the vectors matters less than their direction). $d(x, y) = 1 - \frac{x \cdot y}{\|x\|\|y\|}$

See the earlier lecture for additional options.

Implementation Considerations:

The choice of distance metric should align with the nature of your data. For example, Euclidean distance works well for continuous data, while Hamming distance is more suitable for categorical data.

The next element to consider is the value of k: how many nearest neighbors will be considered?
Choosing the Value of 'K':

- The parameter 'K' determines the number of neighbors to consider.
    - *Low 'K' value (e.g., K = 1):* Can lead to overfitting, as the model might be too sensitive to noise in the training data.
    - *High 'K' value (e.g., K = total number of samples):* Can lead to underfitting, as the model becomes too generalized.
- *Common practice***:** Use cross-validation to select an optimal value for 'K'.

Because of tie-breaking methods, it's common, particularly in binary classification, to avoid even values of k to avoid instability in the model. However, when classification is not binary, choosing the best value of k is more flexible, since ties can happen with any value of k except 1.

Impact of Scaling:

- Distance metrics are sensitive to the scale of the data. Features with larger ranges will dominate the distance calculations.
- ***Standardization/Normalization***:
    - *Standardization*: Transform features to have a mean of 0 and a standard deviation of 1.
    - *Normalization*: Transform features to fall within a specific range, typically [0, 1].

Practical Consideration: Before applying KNN, always scale your features to avoid bias in distance calculation.

Scaling variables (however they are scaled, it should be consistent) can have a huge impact on the quality of your model. It's fine to try it without scaling first, but always check scaling to look for improvement.

Classification (using packages)

```
# Example classification with K = 3
knn_classification <- function(train_data, test_point, k = 3) {
  distances <- sqrt(rowSums((train_data[, -1] - test_point) ^ 2))
  neighbors <- train_data[order(distances), ][1:k, ]
  prediction <- names(sort(table(neighbors[, 1]), decreasing = TRUE))[1]
  return(prediction)
}
```

(This is a generic example. We'll look at the algorithm and an example in more detail.)

Regression (using packages)

```
# Example regression with K = 3
knn_regression <- function(train_data, test_point, k = 3) {
  distances <- sqrt(rowSums((train_data[, -1] - test_point) ^ 2))
  neighbors <- train_data[order(distances), ][1:k, ]
  prediction <- mean(neighbors[, 1])
  return(prediction)
}
```

The main difference between regression and classification is in the final step of the algorithm. In classification, the nearest neighbors “vote” on which class the new element belongs to, with the one with the most votes winning (subject to potential tiebreaking). In regression, the nearest neighbors are averaged together to get the regression prediction.

Advantages of KNN:

- *Simplicity*: Easy to understand and implement.
- *Versatility:* Can be used for classification and regression.
- *No Training Phase:* Works directly on raw data, which can be advantageous in some scenarios.

Disadvantages:

- *Computationally Expensive:* Particularly with large datasets, as distance calculations must be repeated for each prediction.
- *Memory Intensive:* Stores the entire training dataset.
- *Sensitive to Noise:* Outliers can significantly affect the prediction.

Improvements:

- *KD-Trees or Ball Trees* for efficient neighbor searches.
- *Weighted KNN:* Assign weights to neighbors based on their distance, giving closer neighbors more influence on the prediction.

Let’s start with classification and look at two examples, one where we employ the Euclidean distance metric and a second that uses a different one.

Euclidean distance:

```r
# Prepare data
set.seed(123)
mtcars$am <- as.factor(mtcars$am)  # Convert 'am' to a factor (for classification)

# Use 'hp', 'wt', and 'qsec' as predictors and 'am' as the target
data <- mtcars[, c("hp", "wt", "qsec", "am")]

# Scale the predictors
data[, 1:3] <- scale(data[, 1:3])

# Euclidean distance function
euclidean_distance <- function(x1, x2) {
  sqrt(sum((x1 - x2) ^ 2))
}

# KNN function using Euclidean distance
knn_euclidean <- function(train_data, test_point, k = 3) {
  distances <- apply(train_data[, -ncol(train_data)], 1, function(row) euclidean_distance(row,
test_point))
  sorted_indices <- order(distances)
  nearest_neighbors <- train_data[sorted_indices[1:k], ]
  # Return the majority vote for classification
  prediction <- names(sort(table(nearest_neighbors[, ncol(nearest_neighbors)]), decreasing =
TRUE))[1]

  return(as.numeric(prediction))
}

# Split the data into train and test sets
train_data <- data[-1, ]  # Use all rows except the first as training data
test_point <- data[1, -4]  # Use the first row (without the target column) as the test point

# Predict using KNN with Euclidean distance
predicted_value <- knn_euclidean(train_data, test_point, k = 3)
predicted_value
```

For the second case, let's try cosine similarity.

```
# Cosine similarity function
cosine_similarity <- function(x1, x2) {
  sum(x1 * x2) / (sqrt(sum(x1 ^ 2)) * sqrt(sum(x2 ^ 2)))
}

# Split the data into train and test sets
train_data <- data[-1, ]  # Use all rows except the first as training data
test_point <- data[1, -4]  # Use the first row (without the target column) as the test point

# Predict using KNN with Euclidean distance
predicted_value <- knn_euclidean(train_data, test_point, k = 3)
predicted_value

# KNN function using Cosine Similarity
knn_cosine <- function(train_data, test_point, k = 3) {
  similarities <- apply(train_data[, -ncol(train_data)], 1, function(row) cosine_similarity(row,
test_point))
  sorted_indices <- order(-similarities)  # Sort in decreasing order because higher cosine similarity
is better
  nearest_neighbors <- train_data[sorted_indices[1:k], ]

  # Return the majority vote for classification
  prediction <- names(sort(table(nearest_neighbors[, ncol(nearest_neighbors)]), decreasing =
TRUE))[1]

  return(as.numeric(prediction))
}

# Predict using KNN with Cosine similarity
predicted_value_cosine <- knn_cosine(train_data, test_point, k = 3)
predicted_value_cosine
```

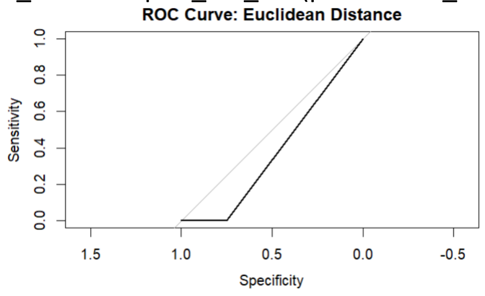
In this example, the first row of mtcars was removed as a test point, and the model was built on the other available datapoints. Both methods of distance metrics correctly predicted the value of 1 for the am variable (compare with the first row of mtcars for verification).

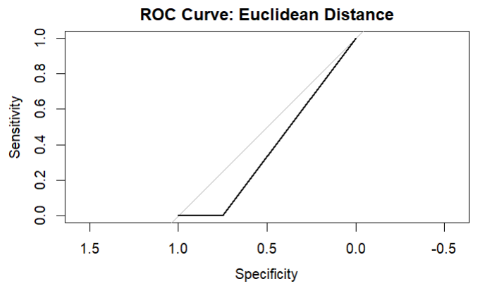
We can also include metrics for comparison if we tweak this a bit. ROC and AUC are most easily done on binary problems, so let's look at those first, before we move on a case with three or more classes.

```
# Load the mtcars dataset
data(mtcars)

# Convert 'am' to a factor for classification
mtcars$am <- as.factor(mtcars$am)

# Select the numeric predictor variables and normalize them
mtcars[, c("mpg", "hp")] <- scale(mtcars[, c("mpg", "hp")])

# Split the data into training and testing sets (70% train, 30% test)
set.seed(123)
train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[train_indices, c("mpg", "hp", "am")]
test_data <- mtcars[-train_indices, c("mpg", "hp", "am")]

euclidean_distance <- function(x1, x2) {
  sqrt(sum((as.numeric(x1) - as.numeric(x2)) ^ 2))
}

cosine_similarity <- function(x1, x2) {
  sum(as.numeric(x1) * as.numeric(x2)) / (sqrt(sum(as.numeric(x1)^2)) *
sqrt(sum(as.numeric(x2)^2)))
}

knn_classifier <- function(train_data, test_point, k = 3, distance_func) {
  distances <- apply(train_data[, -ncol(train_data)], 1, function(row) distance_func(row,
test_point))

  # Determine whether to sort distances ascending or descending
  if (identical(distance_func, cosine_similarity)) {
    nearest_neighbors <- order(distances, decreasing = TRUE)
  } else {
    nearest_neighbors <- order(distances)
  }

  nearest_neighbors <- train_data[nearest_neighbors[1:k], "am"]

  # Majority voting
  prediction <- as.character(sort(table(nearest_neighbors), decreasing = TRUE)[1])
  return(prediction)
}
```

```
predict_knn <- function(train_data, test_data, k = 3, distance_func) {
  predictions <- sapply(1:nrow(test_data), function(i) {
    knn_classifier(train_data, test_data[i, -ncol(test_data)], k, distance_func)
  })
  return(as.factor(predictions))
}

# Predictions using Euclidean distance
predictions_euclidean <- predict_knn(train_data, test_data, k = 3, distance_func =
euclidean_distance)

# Predictions using Cosine similarity
predictions_cosine <- predict_knn(train_data, test_data, k = 3, distance_func = cosine_similarity)

library(caret)

evaluate_model <- function(predictions, true_labels) {
  confusion <- confusionMatrix(predictions, true_labels)
  accuracy <- confusion$overall["Accuracy"]
  f1 <- confusion$byClass["F1"]

  return(list(Confusion_Matrix = confusion$table, Accuracy = accuracy, F1_Score = f1))
}

library(caret)

evaluate_model <- function(predictions, true_labels) {
  # Ensure that predictions and true labels have the same levels
  levels(predictions) <- levels(true_labels)

  # Calculate confusion matrix, accuracy, and F1 score
  confusion <- confusionMatrix(predictions, true_labels)
  accuracy <- confusion$overall["Accuracy"]
  f1 <- confusion$byClass["F1"]

  return(list(Confusion_Matrix = confusion$table, Accuracy = accuracy, F1_Score = f1))
}

# Euclidean results
euclidean_results <- evaluate_model(predictions_euclidean, test_data$am)
print("Euclidean Results")
print(euclidean_results)

# Cosine results
cosine_results <- evaluate_model(predictions_cosine, test_data$am)
print("Cosine Results")
print(cosine_results)
```

```
library(pROC)

plot_roc_auc <- function(predictions, true_labels, title) {
  roc_obj <- roc(as.numeric(true_labels) - 1, as.numeric(predictions) - 1)
  auc_value <- auc(roc_obj)

  plot(roc_obj, main = paste("ROC Curve:", title))
  return(auc_value)
}

# ROC and AUC for Euclidean
auc_euclidean <- plot_roc_auc(predictions_euclidean, test_data$am, "Euclidean Distance")

# ROC and AUC for Cosine
auc_cosine <- plot_roc_auc(predictions_cosine, test_data$am, "Cosine Similarity")
```

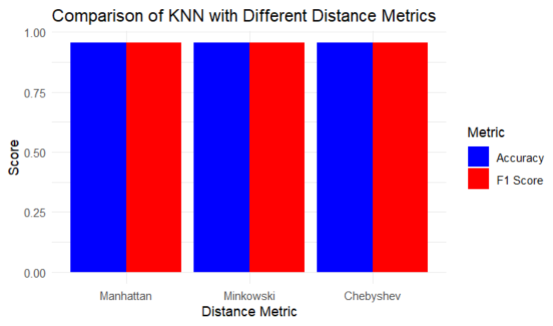




Let's look at an example that generates some classification metrics for us, so that we can compare how well various distance metrics work. We'll also use the iris dataset which has three classes to work with.

```
# Load the dataset
data(iris)
set.seed(123)

# Convert species to a factor (for classification)
iris$Species <- as.factor(iris$Species)

# Scale the predictors
iris[, 1:4] <- scale(iris[, 1:4])

# Split the data into training and testing sets (70% train, 30% test)
train_indices <- sample(1:nrow(iris), 0.7 * nrow(iris))
train_data <- iris[train_indices, ]
test_data <- iris[-train_indices, ]

# Manhattan distance function
manhattan_distance <- function(x1, x2) {
  sum(abs(x1 - x2))
}
```

```r
# Minkowski distance function with p = 3
minkowski_distance <- function(x1, x2, p = 3) {
  sum(abs(x1 - x2) ^ p)^(1/p)
}

# Chebyshev distance function
chebyshev_distance <- function(x1, x2) {
  max(abs(x1 - x2))
}

# General KNN function
knn_general <- function(train_data, test_point, k = 3, distance_func) {
  distances <- apply(train_data[, -ncol(train_data)], 1, function(row) distance_func(row,
test_point))
  sorted_indices <- order(distances)
  nearest_neighbors <- train_data[sorted_indices[1:k], ]

  # Return the majority vote for classification
  prediction <- names(sort(table(nearest_neighbors[, ncol(nearest_neighbors)]), decreasing =
TRUE))[1]

  return(prediction)
}

# Function to calculate evaluation metrics
evaluate_knn <- function(train_data, test_data, k, distance_func) {
  predictions <- sapply(1:nrow(test_data), function(i) {
    knn_general(train_data, test_data[i, -ncol(test_data)], k, distance_func)
  })

  true_labels <- test_data$Species

  # Calculate confusion matrix
  confusion_matrix <- table(Predicted = predictions, Actual = true_labels)

  # Calculate accuracy
  accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)

  # Calculate F1 score (macro-averaged)
  f1_score <- mean(sapply(levels(true_labels), function(class) {
    precision <- confusion_matrix[class, class] / sum(confusion_matrix[class, ])
    recall <- confusion_matrix[class, class] / sum(confusion_matrix[, class])
    f1 <- ifelse(precision + recall > 0, 2 * precision * recall / (precision + recall), 0)
    return(f1)
  }))

  list(predictions = predictions, accuracy = accuracy, f1_score = f1_score, confusion_matrix =
confusion_matrix)
}

# Evaluate with Manhattan Distance
results_manhattan <- evaluate_knn(train_data, test_data, k = 5, distance_func =
manhattan_distance)
results_manhattan$accuracy
results_manhattan$f1_score
results_manhattan$confusion_matrix

# Evaluate with Minkowski Distance (p = 3)
results_minkowski <- evaluate_knn(train_data, test_data, k = 5, distance_func = function(x1, x2)
minkowski_distance(x1, x2, p = 3))
results_minkowski$accuracy
results_minkowski$f1_score
results_minkowski$confusion_matrix

# Evaluate with Chebyshev Distance
results_chebyshev <- evaluate_knn(train_data, test_data, k = 5, distance_func =
chebyshev_distance)
results_chebyshev$accuracy
results_chebyshev$f1_score
results_chebyshev$confusion_matrix

# Prepare data for plotting
plot_data <- data.frame(
  Metric = rep(c("Accuracy", "F1 Score"), each = 3),
  Value = c(results_manhattan$accuracy, results_minkowski$accuracy,
results_chebyshev$accuracy,
          results_manhattan$f1_score, results_minkowski$f1_score, results_chebyshev$f1_score),
  Distance = factor(rep(c("Manhattan", "Minkowski", "Chebyshev"), 2), levels = c("Manhattan",
"Minkowski", "Chebyshev"))
)

# Plot the results
library(ggplot2)

ggplot(plot_data, aes(x = Distance, y = Value, fill = Metric)) +
  geom_bar(stat = "identity", position = "dodge") +
  theme_minimal() +
  labs(title = "Comparison of KNN with Different Distance Metrics", y = "Score", x = "Distance
Metric") +
  scale_fill_manual(values = c("Accuracy" = "blue", "F1 Score" = "red"))
```



As it turns out here, the metrics turn out largely the same for all three distance metrics, but there is a slight difference in one of the confusion matrices. The important thing to keep in mind is that you can't know which distance metric will produce the best results (or the same results) unless you test them.

Resources:

1. https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html
2. https://www.analyticsvidhya.com/blog/2021/08/how-knn-uses-distance-measures/
3. https://medium.com/@luigi.fiori.lf0303/distance-metrics-and-k-nearest-neighbor-knn-1b840969c0f4
4. https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d
5. https://www.linkedin.com/advice/3/what-most-effective-distance-metrics-optimizing-xndwc