005921309_stats102b_hw5

Anish Deshpande

2024-06-07

Question 1:

a.)

```r
# read in pokemon data
pokemon_data <- read_csv("pokemon.csv", show_col_types = FALSE)
# Find the index of the first and last column to standardize
start_col <- which(names(pokemon_data) == "height_m")
end_col <- which(names(pokemon_data) == "base_experience")

# Standardize the columns from height_m to base_experience
pokemon_data[, start_col:end_col] <- lapply(pokemon_data[, start_col:end_col], scale)


# subset pokemon data by generation
gen1 <- pokemon_data %>% filter(generation == 1)
gen2 <- pokemon_data %>% filter(generation == 2)
# show first 6 rows of gen1:
head(gen1)
```

```
## # A tibble: 6 x 17
##   name      generation type_1 type_2 height_m[,1] weight_kg[,1] total_points[,1]
##   <chr>          <dbl> <chr>  <chr>         <dbl>         <dbl>            <dbl>
## 1 Bulbasaur          1 Grass  Poison       -0.433        -0.497           -0.999
## 2 Ivysaur            1 Grass  Poison       -0.203        -0.448           -0.274
## 3 Venusaur           1 Grass  Poison        0.566         0.251            0.726
## 4 Mega Ven~          1 Grass  Poison        0.873         0.696            1.56
## 5 Charmand~          1 Fire   <NA>         -0.510        -0.484           -1.07
## 6 Charmele~          1 Fire   <NA>         -0.126        -0.400           -0.274
## # i 10 more variables: hp <dbl[,1]>, attack <dbl[,1]>, defense <dbl[,1]>,
## #   sp_attack <dbl[,1]>, sp_defense <dbl[,1]>, speed <dbl[,1]>,
## #   catch_rate <dbl[,1]>, base_friendship <dbl[,1]>, base_experience <dbl[,1]>,
## #   is_legendary <lgl>
```

```r
# show first 6 rows of gen2:
head(gen2)
```

```
## # A tibble: 6 x 17
##   name      generation type_1 type_2 height_m[,1] weight_kg[,1] total_points[,1]
##   <chr>          <dbl> <chr>  <chr>         <dbl>         <dbl>             <dbl>
## 1 Chikorita          2 Grass  <NA>        -0.279        -0.501            -0.999
## 2 Bayleef            2 Grass  <NA>        -0.0489       -0.425            -0.274
## 3 Meganium           2 Grass  <NA>         0.412         0.255             0.726
## 4 Cyndaquil          2 Fire   <NA>        -0.587        -0.489            -1.07
## 5 Quilava            2 Fire   <NA>        -0.279        -0.400            -0.274
## 6 Typhlosi~          2 Fire   <NA>         0.335         0.0861            0.801
## # i 10 more variables: hp <dbl[,1]>, attack <dbl[,1]>, defense <dbl[,1]>,
## #   sp_attack <dbl[,1]>, sp_defense <dbl[,1]>, speed <dbl[,1]>,
## #   catch_rate <dbl[,1]>, base_friendship <dbl[,1]>, base_experience <dbl[,1]>,
## #   is_legendary <lgl>
```

b.)

```r
# trainx: a data frame of the training observations
# trainy: a vector of the training labels
# testx: a data frame of the testing (new) observations
# k: the number of neighbors
# output: a vector of the predicted labels corresponding to the new observations

# first write helper function to get Euclidean Distance given a vector x and y of equal length:
GetEuclidianDistance <- function(x, y) {
  result <- sqrt(sum((x - y)^2))
  # return the result
  result
}

# knn function implementation:
knn <- function(trainx, trainy, testx, k) {
  # set result to be empty vector:
  result <- c()

  # for each new testx case:
  for (i in 1:nrow(testx)) {
    # create empty distances vector
    distances <- c()
    # compute the distance between the test case and each training observation:
    for (j in 1:nrow(trainx)) {
      distance <- GetEuclidianDistance(testx[i, ], trainx[j, ])
      # append distance to distance vector
      distances <- append(distances, distance)
    }
    # create a data frame of distances and indexes, then arrange by increasing distance
    indexes <- c(1:length(trainy))
    point_data_frame <- data.frame(distances = distances, indexes = indexes) %>% arrange(distances)
    # choose the k nearest neighbors:
    interested_indexes <- point_data_frame$indexes[1:k]
    # get the extracted y labels:
    nearest_labels <- trainy[interested_indexes]
    freq_table <- table(nearest_labels)
    label <- names(freq_table[which.max(freq_table)])
    # append label to the result
    result <- append(result, label)
  }
  # convert result to logical
  result <- as.logical(result)
  # return the result
  result
}
```

c.)

```
trainx <- gen1[, c("total_points", "hp")]
testx <- gen2[, c("total_points", "hp")]
trainy <- gen1$is_legendary

knn(trainx, trainy, testx, 5)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [97]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

```
# compute the misclassification rate...
# creating a helper function:
GetMisclassificationRate <- function(predicted_labels, actual_labels) {
  correct <- 0
  wrong <- 0
  # for each label, check if it is correctly classified:
  for (i in 1:length(actual_labels)) {
    if (predicted_labels[i] == actual_labels[i]) {
      correct <- correct + 1
    } else {
      wrong <- wrong + 1
    }
  }
  result <- wrong / (wrong + correct)
  # return the result
  result
}

pred_labels <- knn(trainx, trainy, testx, 5)
true_labels <- gen2$is_legendary

misclassification_rate <- GetMisclassificationRate(pred_labels, true_labels)
misclassification_rate
```

```
## [1] 0.04716981
```

d.)

```r
# set seed for reproducibility:
set.seed(1313)

# use 6 fold cross validation:
k_values <- c(3, 7, 11, 21, 51)

# creating a k fold cross validation function
k_fold_cv <- function(data, k, K = 6) {
  # find number of observations:
  n <- nrow(data)
  # randomly shuffle the indices:
  indices <- sample(1:n)
  # set each testing set to have n/K data points:
  fold_size <- ceiling(n / K)
  # set an empty vector of length K to be filled with misclassification values for later:
  misclassification_values <- numeric(K)

  # for each fold...
  for (i in 1:K) {
    start_index <- (i - 1) * fold_size + 1
    end_index <- min(i * fold_size, n)
    test_indices <- indices[start_index:end_index]
    # set train indices to be everything except what you are testing on
    train_indices <- indices[-(start_index:end_index)]

    # train the model on the training set:
    trainx <- data[train_indices, c("total_points", "hp")]
    trainy <- data[train_indices, ]$is_legendary
    testx <- data[test_indices, c("total_points", "hp")]

    pred_labels <- knn(trainx, trainy, testx, k)
    true_labels <- data[test_indices, ]$is_legendary

    # calculate misclassification rate
    rate <- GetMisclassificationRate(pred_labels, true_labels)
    # append rate to result vector
    misclassification_values <- append(misclassification_values, rate)
  }

  # return the result
  mean(misclassification_values)
}


# run k-fold cross validation for all k values:
cv_results <- sapply(k_values, function(k) k_fold_cv(gen1, k, K = 6))
results_df <- data.frame(k = k_values, misclassification_rate = cv_results)
results_df
```

```
##    k misclassification_rate
## 1  3             0.01344086
## 2  7              0.02419355
## 3 11              0.01881720
## 4 21              0.01881720
## 5 51              0.01881720
```

- From the table above, we see that using kNN with k = 3 is the optimal parameter value.

e.)

```r
# Prepare training and testing data
trainx <- gen1[, c("total_points", "hp")]
trainy <- gen1$is_legendary
testx <- gen2[, c("total_points", "hp")]
testy <- gen2$is_legendary

# Predict using k-NN with k = 3
pred_labels <- knn(trainx, trainy, testx, 3)

# Calculate the misclassification rate
misclassification_rate <- GetMisclassificationRate(pred_labels, testy)
cat("Misclassification Rate:", misclassification_rate, "\n")
```

```
## Misclassification Rate: 0.04716981
```

```r
# Create the confusion matrix
confusion_matrix <- table(Predicted = pred_labels, Actual = testy)

# Display the confusion matrix
confusion_matrix
```

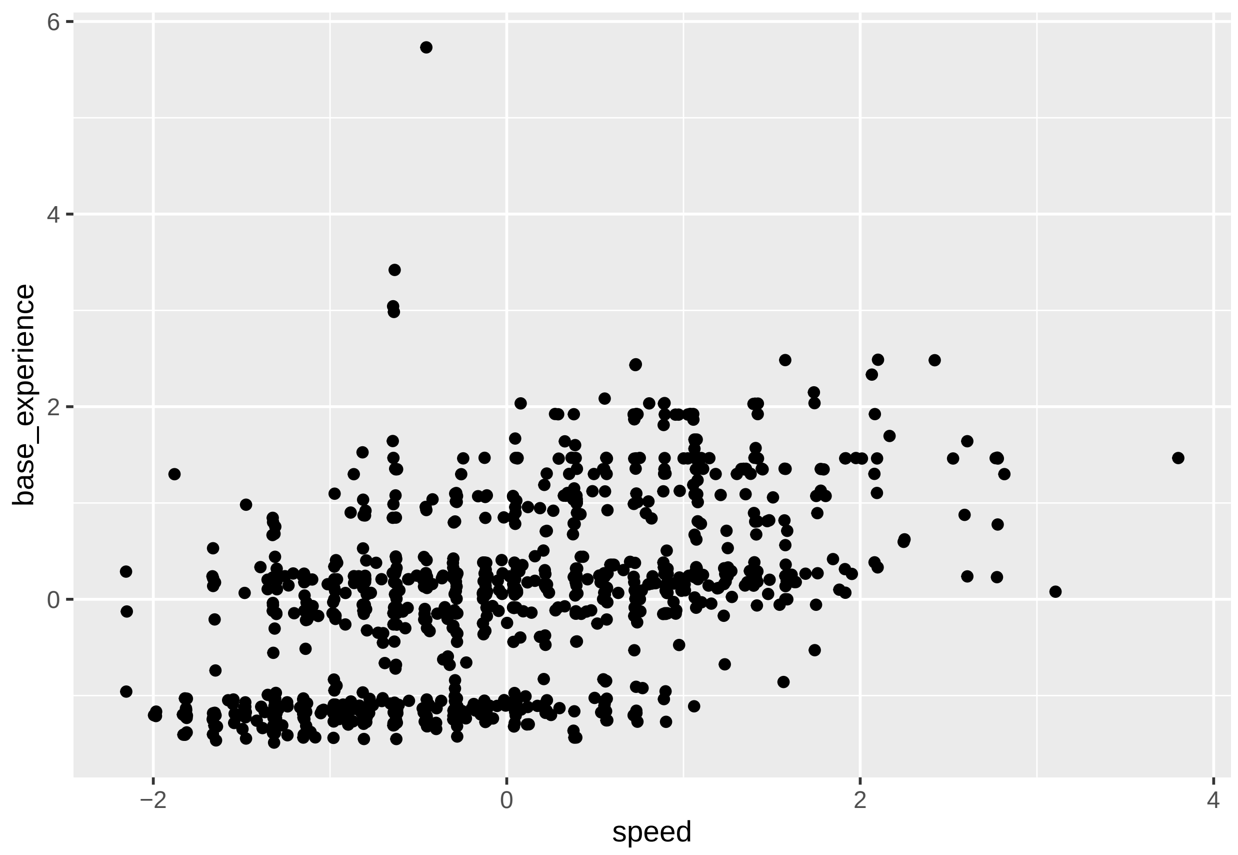
```
##          Actual
## Predicted FALSE TRUE
##     FALSE    98    3
##     TRUE      2    3
```

- The confusion matrix above shows that there were **98** instances in which the algorithm predicted a non-legendary pokemon and was **correct**
- There were **3** instances in which the algorithm predicted a non-legendary pokemon, but was **incorrect**
- There were **2** instances in which we predicted a legendary pokemon, but are **incorrect**
- There were **3** instances where we predicted a legendary pokemon and we are **correct**

Question 2:

a.)

```r
ggplot(data = pokemon_data, aes(x = speed, y = base_experience)) +
  geom_jitter()
```



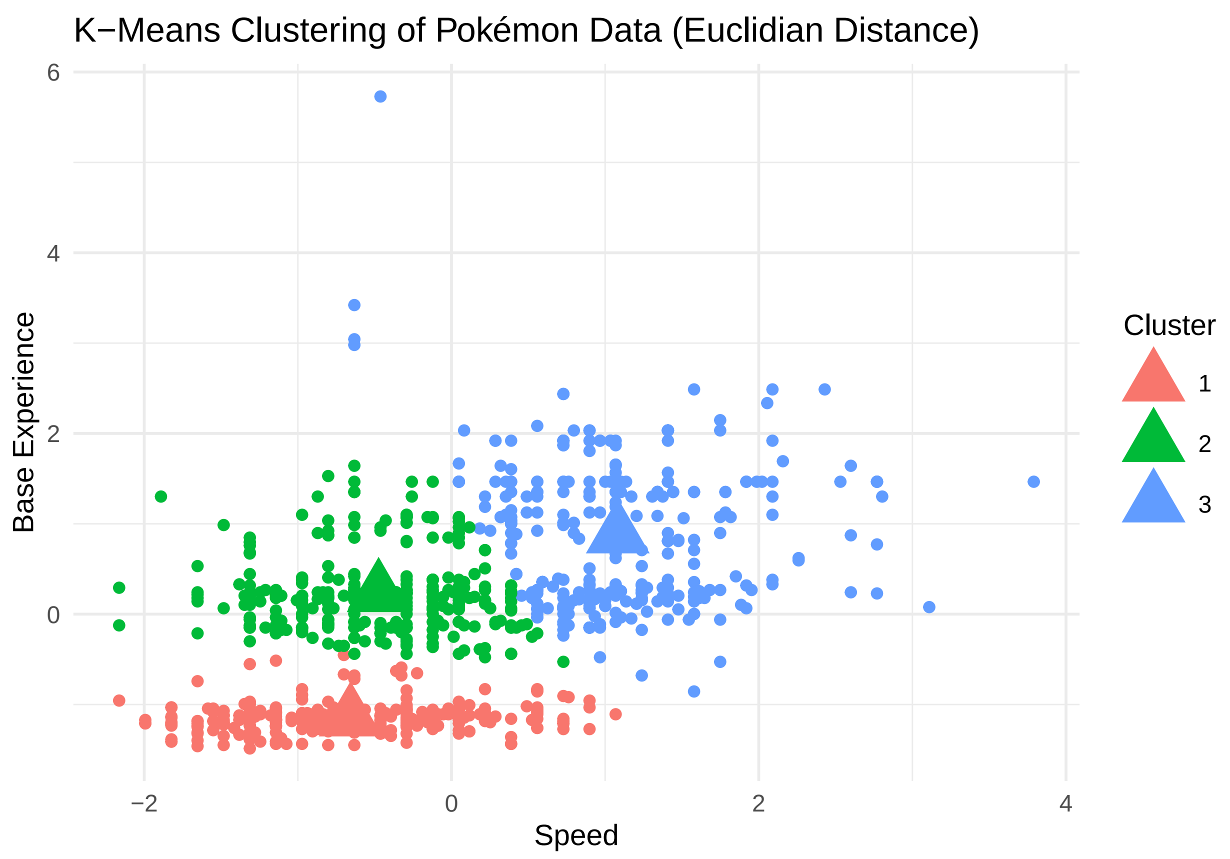
- The scatter plot above shows that there is a positive correlation between speed and base experience, however it is not very strong and cannot be linearly associated. The data seem to form three stacked clusters. k-means clustering can be a useful way of identifying and grouping the data into clusters.

b.)

```
# implementing k-means algorithm...
# trainx is the n observations
# k is the number of clusters
k_means <- function(trainx, k, distance = "euclidean") {
  # Define the distance function based on the specified distance metric
  distance_function <- switch(distance,
                              euclidean = function(a, b) sqrt(sum((a - b)^2)),
                              manhattan = function(a, b) sum(abs(a - b)),
                              mahalanobis = function(a, b)
                                sqrt(as.matrix(a - b) %*% cov_inv %*% as.matrix(t(a - b))),
                              custom = function(a, b)
                                sqrt(as.matrix(a - b) %*% Q %*% as.matrix(t(a - b))),
                              stop("Unsupported distance metric"))
  if (distance == "mahalanobis") {
    # Compute the inverse of the covariance matrix
    cov_mat <- cov(trainx)
    cov_inv <- solve(cov_mat)  # Inverse of the covariance matrix
  }
```

```r
  if (distance == "custom") {
    Q <- matrix(c(1, -2, -2, 10), nrow = 2, ncol = 2, byrow = TRUE)
  }

  # Find the number of observations
  n <- nrow(trainx)
  # Randomly assign a number 1 to k for each of the n observations
  cluster_assignments <- sample(1:k, n, replace = TRUE)
  keep_going <- TRUE

  while (keep_going) {
    keep_going <- FALSE
    centroids <- matrix(0, nrow = k, ncol = ncol(trainx))

    # Compute the centroids for each cluster
    for (i in 1:k) {
      k_data <- trainx[which(cluster_assignments == i), ]
      if (nrow(k_data) > 0) {
        centroids[i, ] <- colMeans(k_data)
      }
    }

    new_assignments <- cluster_assignments

    # Reassign points to the nearest centroid
    for (i in 1:n) {
      distances <- apply(centroids, 1, function(centroid) distance_function(trainx[i, ], centroid))
      new_assignments[i] <- which.min(distances)
    }

    # Check if there are any changes in assignments
    if (!all(new_assignments == cluster_assignments)) {
      keep_going <- TRUE
    }

    cluster_assignments <- new_assignments
  }

  list(assignments = cluster_assignments, centroids = centroids)
}


# run the k-means algorithm with euclidean distance...
trainx <- pokemon_data[, c("speed", "base_experience")]
k_means_result <- k_means(trainx, 3, "euclidean")
trainx$cluster <- as.factor(k_means_result$assignments)

# Convert centroids to a data frame with appropriate column names
centroids_df <- data.frame(k_means_result$centroids)
colnames(centroids_df) <- colnames(trainx)[1:2]
centroids_df$cluster <- as.factor(1:nrow(centroids_df))

# Plotting
```

```r
ggplot(trainx, aes(x = speed, y = base_experience, color = cluster)) +
  geom_point() +
  geom_point(data = centroids_df, aes(x = speed, y = base_experience, color = cluster),
             shape = 17, size = 8) +
  labs(color = "Cluster") +
  theme_minimal() +
  ggtitle("K-Means Clustering of Pokémon Data (Euclidian Distance)") +
  xlab("Speed") +
  ylab("Base Experience")
```

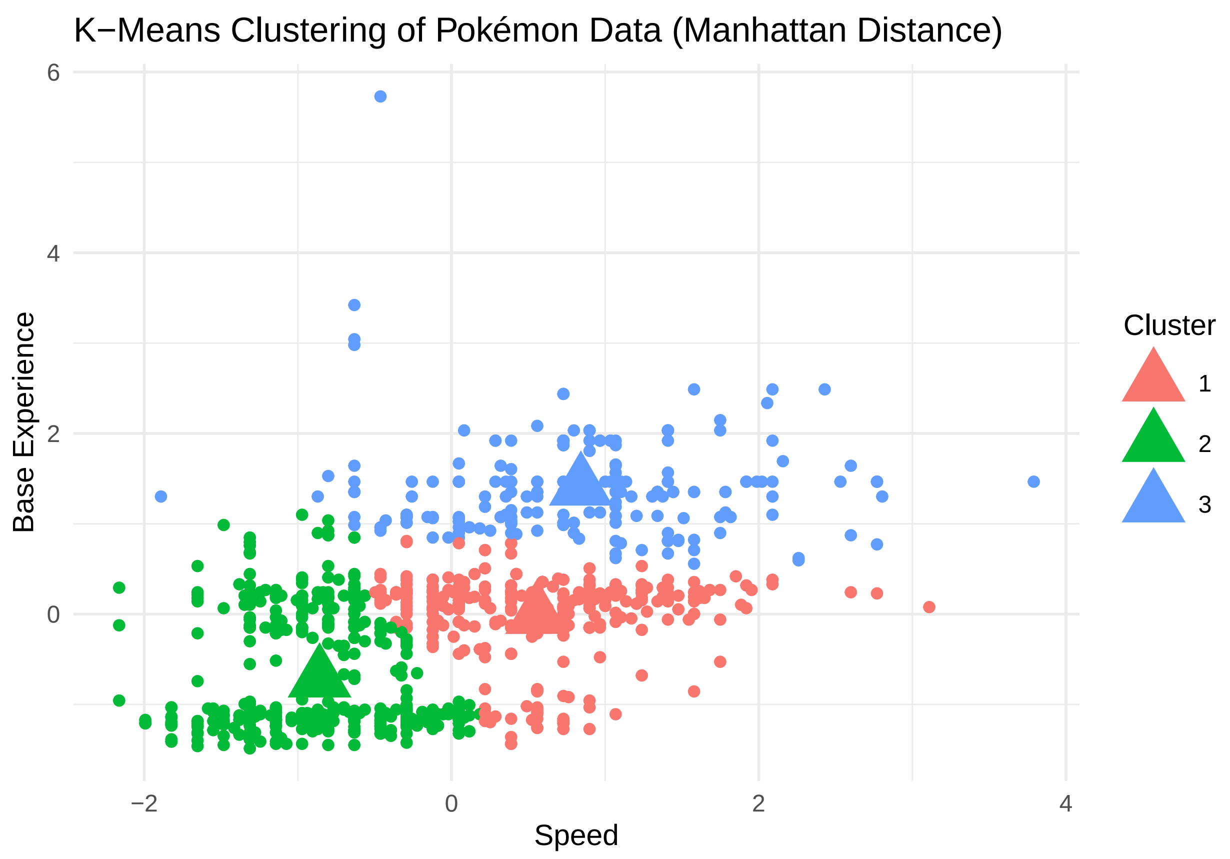


- The euclidian distance is the most commonly used distance metric and is usually suitable for quantitative data. However, euclidean distance applies well when clusters are symmetric and spherical. This is not the case, and you can see how the clusters aren't ideally separated.

c.)

```r
trainx <- pokemon_data[, c("speed", "base_experience")]
k_means_result <- k_means(trainx, 3, "manhattan")
trainx$cluster <- as.factor(k_means_result$assignments)

# Convert centroids to a data frame with appropriate column names
centroids_df <- data.frame(k_means_result$centroids)
colnames(centroids_df) <- colnames(trainx)[1:2]
```

```
centroids_df$cluster <- as.factor(1:nrow(centroids_df))

# Plotting
ggplot(trainx, aes(x = speed, y = base_experience, color = cluster)) +
  geom_point() +
  geom_point(data = centroids_df, aes(x = speed, y = base_experience, color = cluster),
             shape = 17, size = 8) +
  labs(color = "Cluster") +
  theme_minimal() +
  ggtitle("K-Means Clustering of Pokémon Data (Manhattan Distance)") +
  xlab("Speed") +
  ylab("Base Experience")
```

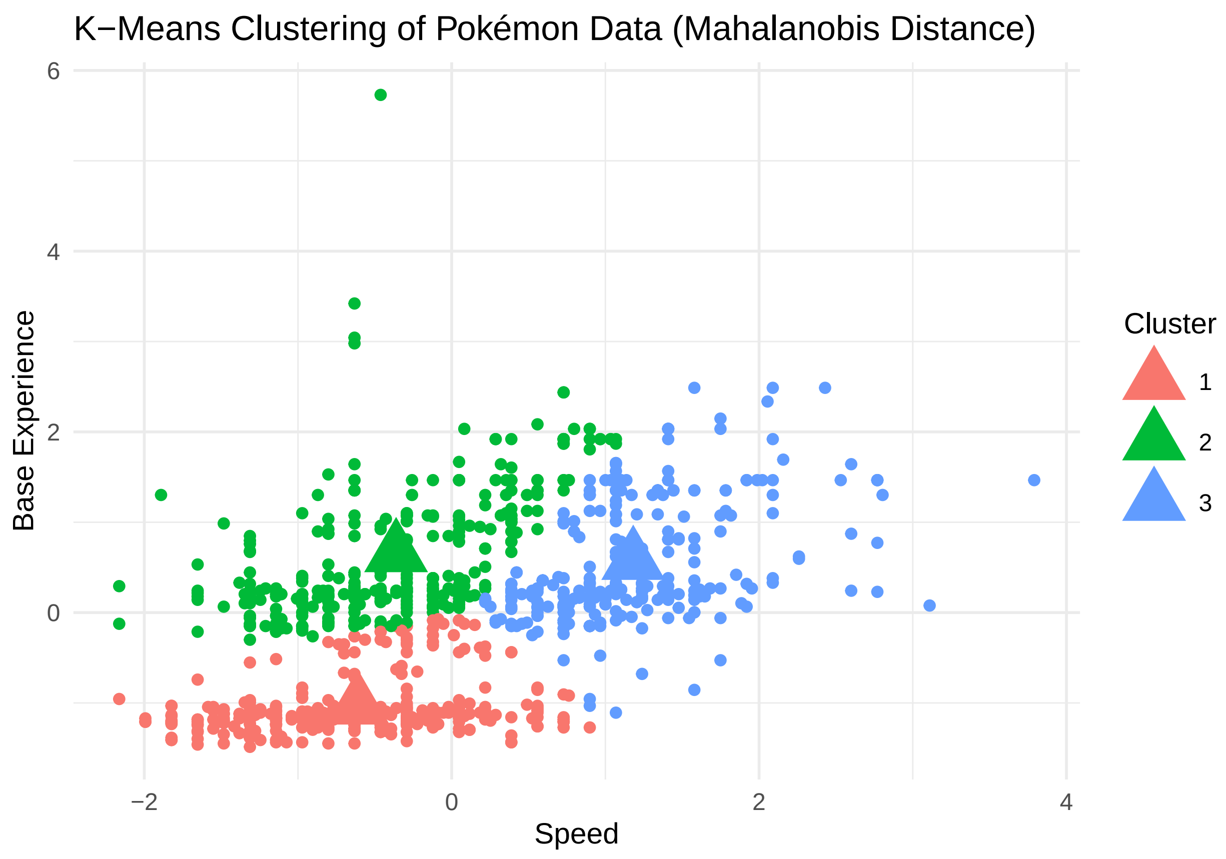


- The manhattan distance is used for quantitative data, but is robust to outliers. This is once again not the best distance metric to use, as it clearly misclassifies some of the clusters. The true clusters seem to be flat bands, but this metric mixes up the clusters completely.

d.)

```
trainx <- pokemon_data[, c("speed", "base_experience")]
k_means_result <- k_means(trainx, 3, "mahalanobis")
trainx$cluster <- as.factor(k_means_result$assignments)
```

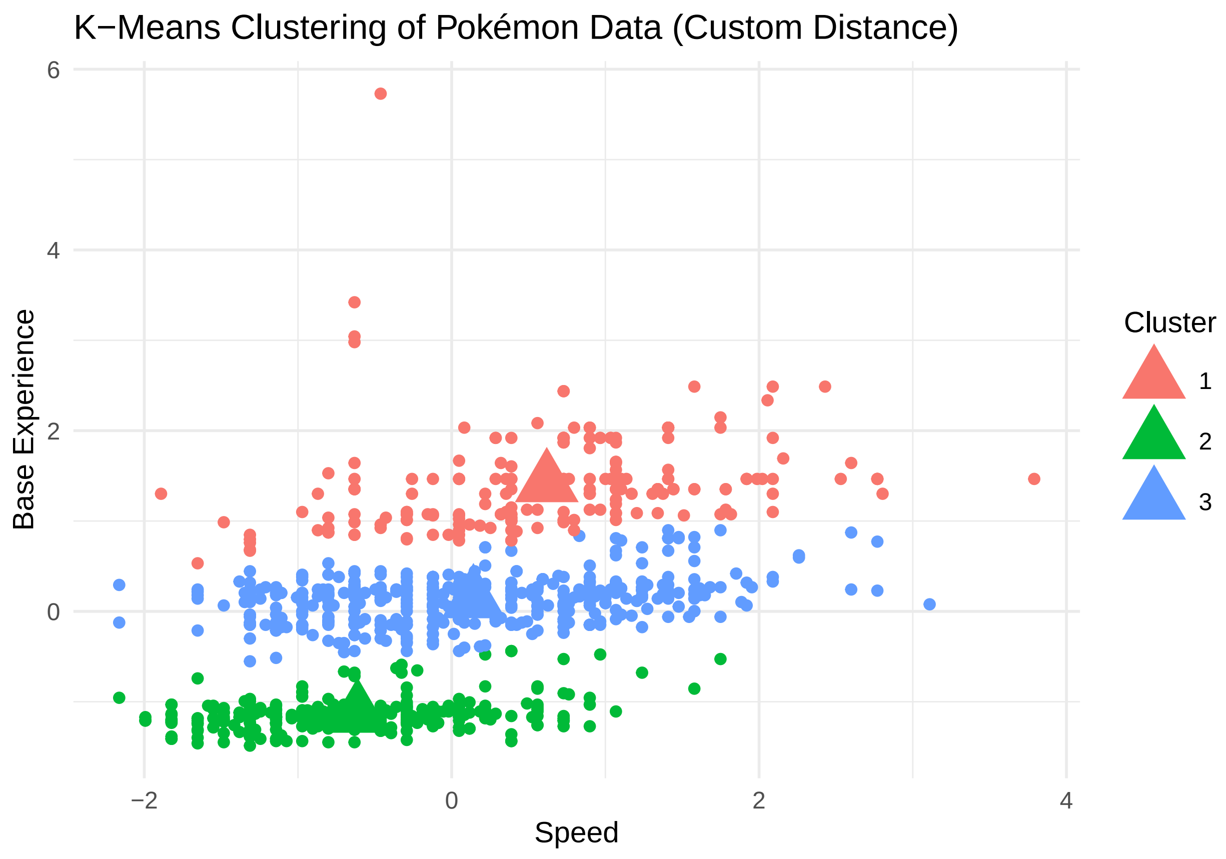
```r
# Convert centroids to a data frame with appropriate column names
centroids_df <- data.frame(k_means_result$centroids)
colnames(centroids_df) <- colnames(trainx)[1:2]
centroids_df$cluster <- as.factor(1:nrow(centroids_df))

# Plotting
ggplot(trainx, aes(x = speed, y = base_experience, color = cluster)) +
  geom_point() +
  geom_point(data = centroids_df, aes(x = speed, y = base_experience, color = cluster),
             shape = 17, size = 8) +
  labs(color = "Cluster") +
  theme_minimal() +
  ggtitle("K-Means Clustering of Pokémon Data (Mahalanobis Distance)") +
  xlab("Speed") +
  ylab("Base Experience")
```



- The mahalanobis distance is suitable for correlated data. This data does look slightly correlated, however the clusters themselves look very horizontal. This distance metric may not be the best, as it tries to create some sort of an association between speed and base experience when forming the three clusters.

e.)

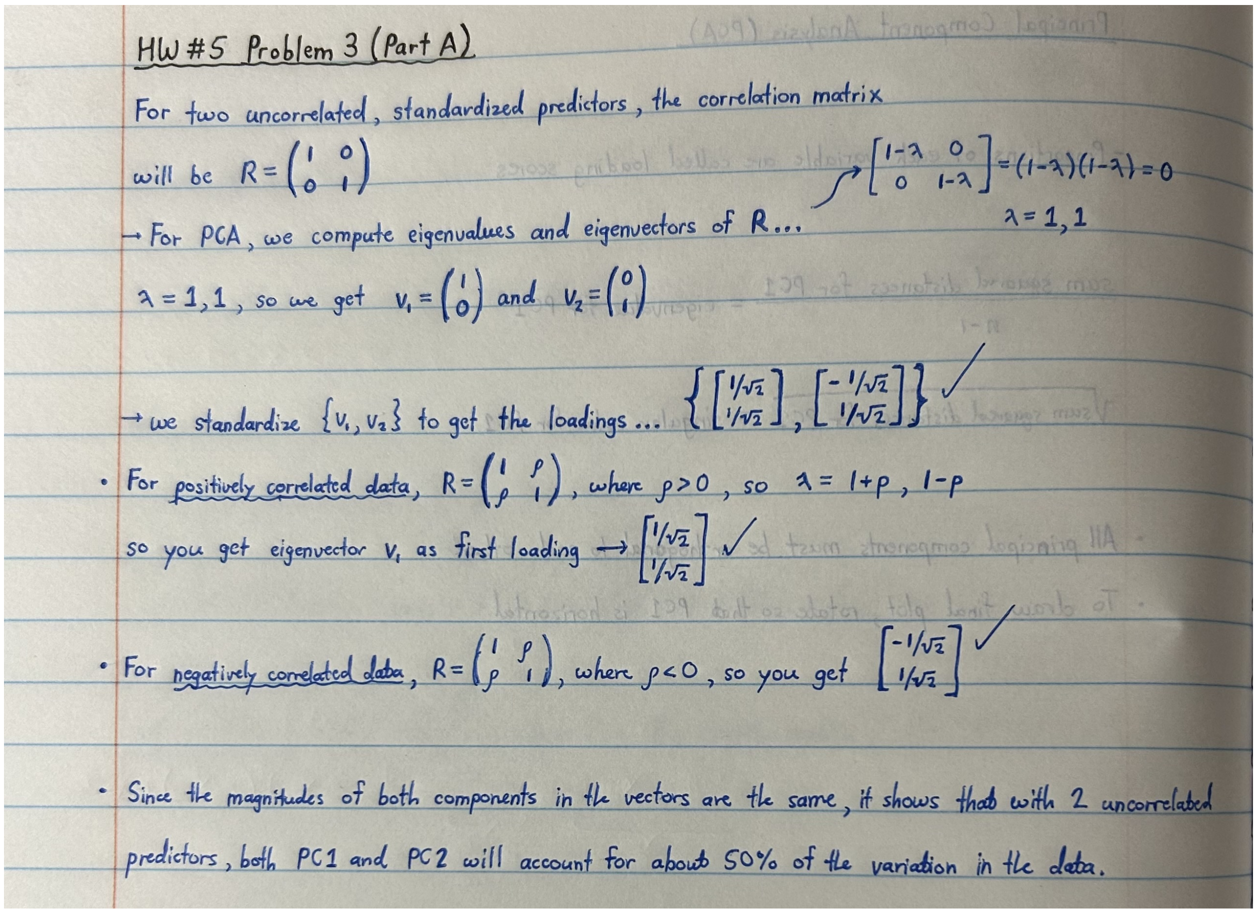
```r
trainx <- pokemon_data[, c("speed", "base_experience")]
k_means_result <- k_means(trainx, 3, "custom")
trainx$cluster <- as.factor(k_means_result$assignments)

# Convert centroids to a data frame with appropriate column names
centroids_df <- data.frame(k_means_result$centroids)
colnames(centroids_df) <- colnames(trainx)[1:2]
centroids_df$cluster <- as.factor(1:nrow(centroids_df))

# Plotting
ggplot(trainx, aes(x = speed, y = base_experience, color = cluster)) +
  geom_point() +
  geom_point(data = centroids_df, aes(x = speed, y = base_experience, color = cluster),
             shape = 17, size = 8) +
  labs(color = "Cluster") +
  theme_minimal() +
  ggtitle("K-Means Clustering of Pokémon Data (Custom Distance)") +
  xlab("Speed") +
  ylab("Base Experience")
```



- The clustering in this plot seems much more elongated and thinned out. This is largely due to the distance metric. If you notice closely, the distance metric used in this model is very similar to the mahalanobis distance, except that we have a matrix Q instead of a covariance matrix. The off diagonal values are both -2, and represent a correlation between the two components of speed and base experience. The top left value, 1, suggests that speed has a standard weight, while the bottom right value, 10, suggests that base experience is much more heavily weighted, resulting in tighter bands for our clusters. This looks like a good distance metric, as it separates out the clusters much to our liking.

Question 3:

a.)



HW #5 Problem 3 (Part A)

For two uncorrelated, standardized predictors, the correlation matrix will be $R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

$\begin{bmatrix} 1-\lambda & 0 \\ 0 & 1-\lambda \end{bmatrix} = (1-\lambda)(1-\lambda) = 0$

$\lambda = 1, 1$

→ For PCA, we compute eigenvalues and eigenvectors of R...

$\lambda = 1, 1$, so we get $v_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $v_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

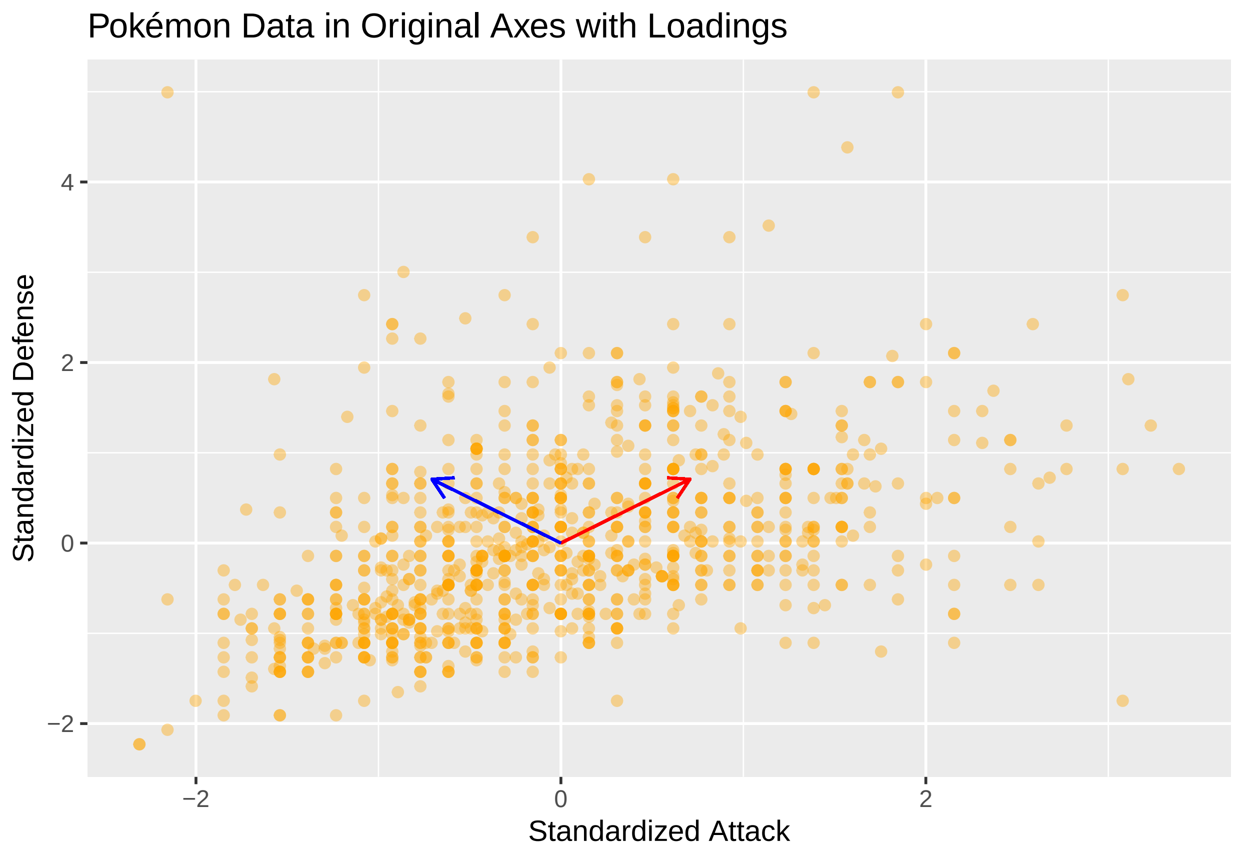
→ we standardize $\{v_1, v_2\}$ to get the loadings ... $\left\{ \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} \right\}$ ✓

- For positively correlated data, $R = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$, where $\rho > 0$, so $\lambda = 1+\rho, 1-\rho$
  so you get eigenvector v_1 as first loading $\rightarrow \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$ ✓
- For negatively correlated data, $R = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$, where $\rho < 0$, so you get $\begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$ ✓
- Since the magnitudes of both components in the vectors are the same, it shows that with 2 uncorrelated predictors, both PC1 and PC2 will account for about 50% of the variation in the data.

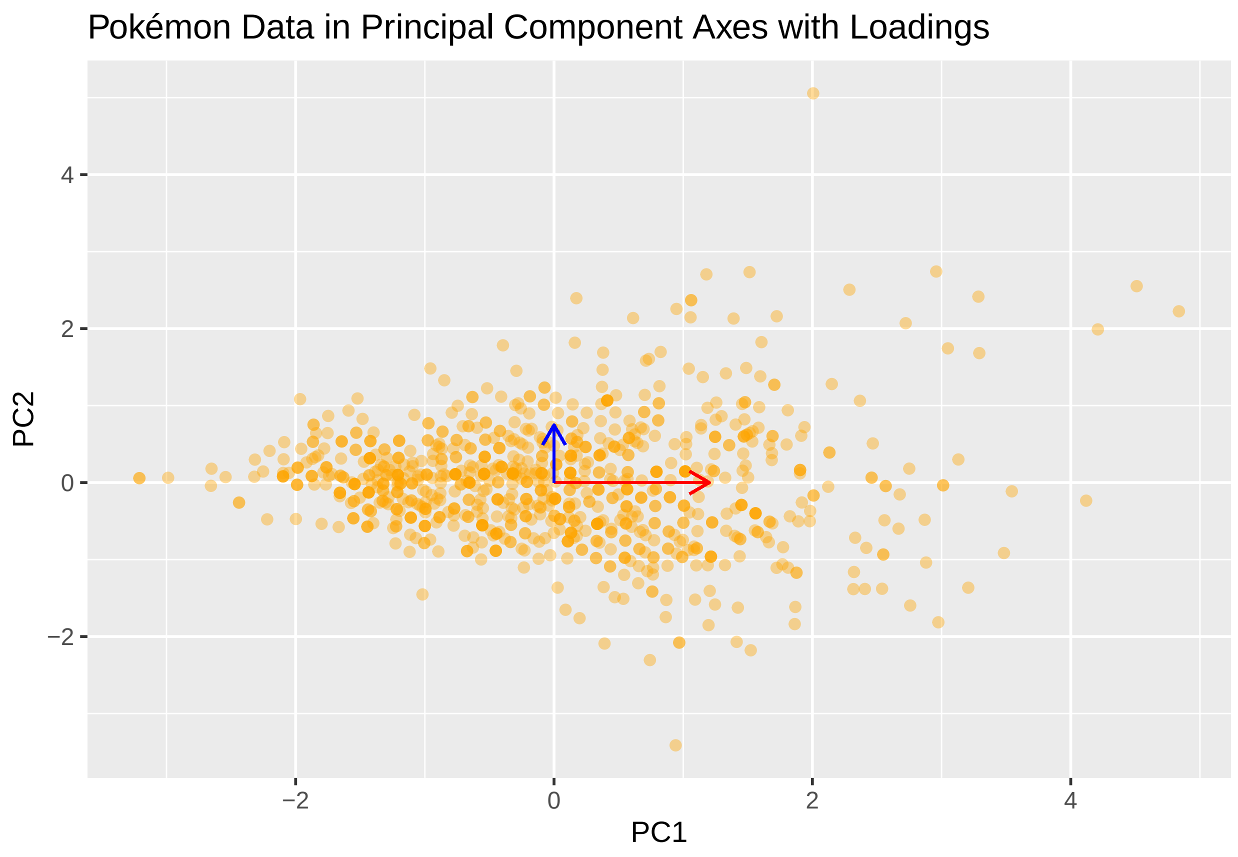
b.)

```r
# standardize attack and defense columns
pokemon_data <- pokemon_data %>%
  mutate(attack_std = scale(attack),
         defense_std = scale(defense))
# extract appropriate columns
data_std <- pokemon_data %>%
  select(c(attack_std, defense_std))
# compute covariance matrix
cov_matrix <- cov(data_std)
```

```r
# get eigenvalues and eigenvectors
eigen_result <- eigen(cov_matrix)
eigenvalues <- eigen_result$values
eigenvectors <- eigen_result$vectors
# project data onto eigenvectors to obtain principal components
principal_components <- as.matrix(data_std) %*% eigenvectors
# data frame for plot...
data_std_df <- as.data.frame(data_std)
colnames(data_std_df) <- c("attack", "defense")
principal_components_df <- as.data.frame(principal_components)
colnames(principal_components_df) <- c("PC1", "PC2")

# plot in original axes:
ggplot(data_std_df, aes(x = attack, y = defense)) +
  geom_point(color = 'orange', alpha = 0.4) +
  geom_segment(aes(x = 0, y = 0, xend = eigenvectors[1,1], yend = eigenvectors[2,1]),
               arrow = arrow(length = unit(0.3, "cm")), color = "red") +
  geom_segment(aes(x = 0, y = 0, xend = eigenvectors[1,2], yend = eigenvectors[2,2]),
               arrow = arrow(length = unit(0.3, "cm")), color = "blue") +
  ggtitle("Pokémon Data in Original Axes with Loadings") +
  xlab("Standardized Attack") + ylab("Standardized Defense")
```



```r
# plot in principal component axes:
ggplot(principal_components_df, aes(x = PC1, y = PC2)) +
  geom_point(color = 'orange', alpha = 0.4) +
  geom_segment(aes(x = 0, y = 0, xend = sqrt(eigenvalues[1]), yend = 0),
               arrow = arrow(length = unit(0.3, "cm")), color = "red") +
  geom_segment(aes(x = 0, y = 0, xend = 0, yend = sqrt(eigenvalues[2])),
               arrow = arrow(length = unit(0.3, "cm")), color = "blue") +
  ggtitle("Pokémon Data in Principal Component Axes with Loadings") +
  xlab("PC1") + ylab("PC2")
```



c.)

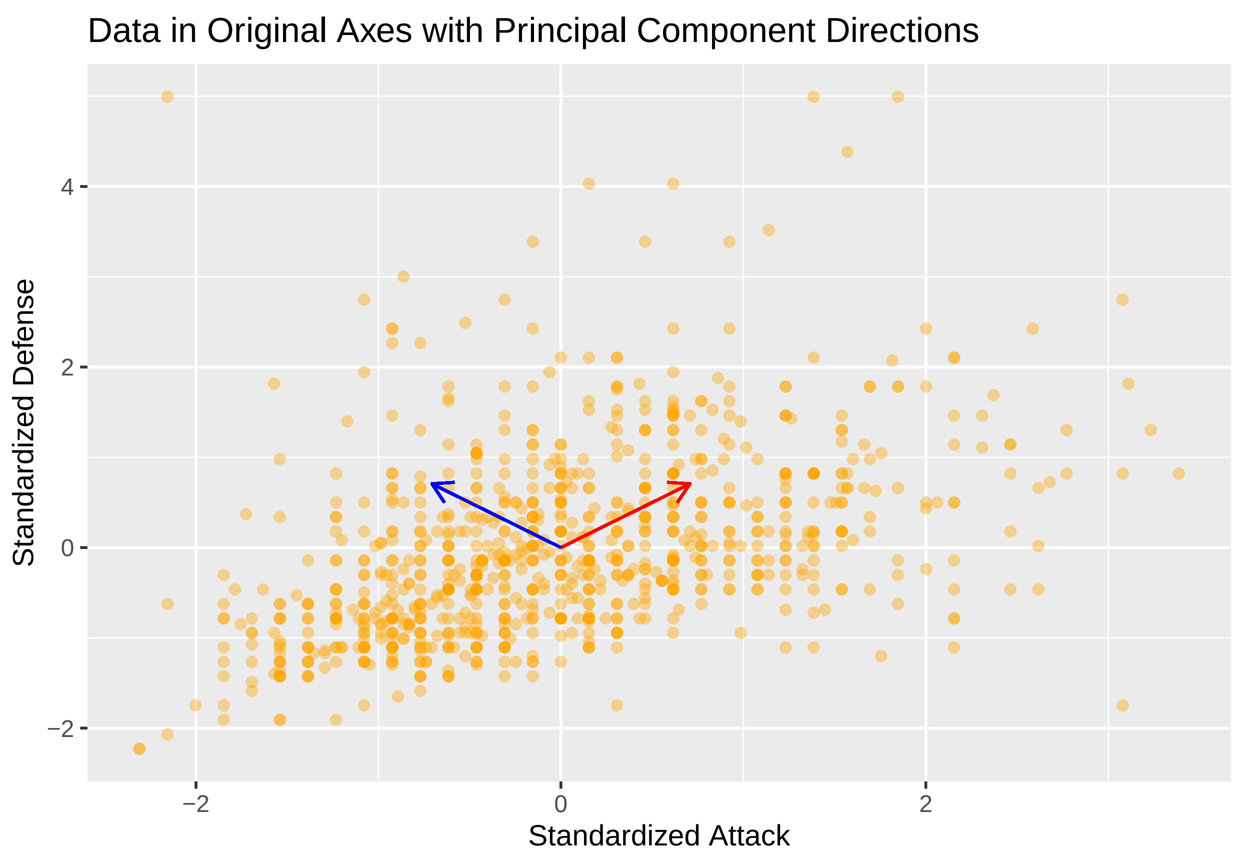
```r
# Select the numeric columns from height_m to base_experience
numeric_columns <- pokemon_data %>%
  select(height_m:base_experience)
# Standardize the selected columns
numeric_columns_std <- scale(numeric_columns)
# get covariance matrix
cov_matrix <- cov(numeric_columns_std)
# get eigenvalues and eigenvectors
eigen_result <- eigen(cov_matrix)
eigenvalues <- eigen_result$values
eigenvectors <- eigen_result$vectors
# project data onto eigenvectors to obtain principal components
principal_components <- as.matrix(numeric_columns_std) %*% eigenvectors
```

```r
# print the result:
head(principal_components[, 1])
```

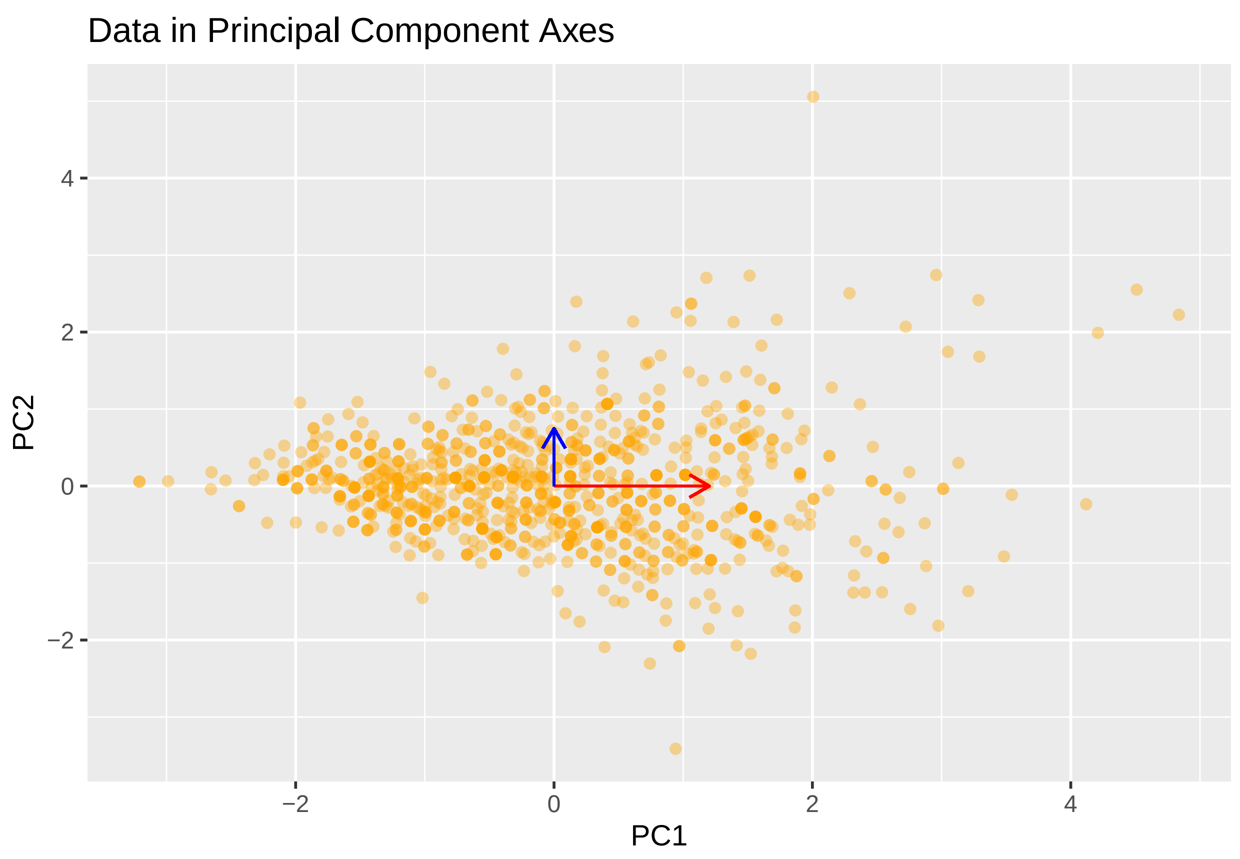
```
## [1]  1.9677574  0.4519699 -1.8521226 -3.4846844  2.1705764  0.4797547
```

d.)

```r
# This is for checking part B:
# perform PCA using prcomp() function...
pca_result <- prcomp(data_std, center = TRUE, scale. = TRUE)
# Extract the loadings (principal component directions)
loadings <- pca_result$rotation
# Extract the principal components
pca_data <- as.data.frame(pca_result$x)

# Original data plot with loading vectors
ggplot(pokemon_data, aes(x = attack_std, y = defense_std)) +
  geom_point(color = 'orange', alpha = 0.4) +
  geom_segment(aes(x = 0, y = 0, xend = loadings[1, 1], yend = loadings[2, 1]),
               arrow = arrow(length = unit(0.3, "cm")), color = 'red') +
  geom_segment(aes(x = 0, y = 0, xend = loadings[1, 2], yend = loadings[2, 2]),
               arrow = arrow(length = unit(0.3, "cm")), color = 'blue') +
  ggtitle("Data in Original Axes with Principal Component Directions") +
  xlab("Standardized Attack") +
  ylab("Standardized Defense")
```



```r
# PCA data plot with loading vectors
ggplot(pca_data, aes(x = PC1, y = PC2)) +
  geom_point(color = 'orange', alpha = 0.4) +
  geom_segment(aes(x = 0, y = 0, xend = pca_result$sdev[1], yend = 0),
               arrow = arrow(length = unit(0.3, "cm")), color = 'red') +
  geom_segment(aes(x = 0, y = 0, xend = 0, yend = pca_result$sdev[2]),
               arrow = arrow(length = unit(0.3, "cm")), color = 'blue') +
  ggtitle("Data in Principal Component Axes") +
  xlab("PC1") +
  ylab("PC2")
```



```r
# this is for checking part C:
# Perform PCA on the standardized data
pca_result <- prcomp(numeric_columns_std, center = TRUE, scale. = TRUE)
# Extract the first principal component
first_principal_component <- pca_result$x[, 1]
# Display the first 6 elements of the first principal component
head(first_principal_component)
```

```
## [1] -1.9677574 -0.4519699  1.8521226  3.4846844 -2.1705764 -0.4797547
```

- Notice that the principal component values are the negative of what I got in part C. This doesn't matter, since this just means that the vector points in the exact opposite direction. The principal component essentially behaves the same.