Train and Evaluate Clustering Models using Tidymodels and friends

Eric Wanjau

Clustering Challenge

Clustering is an *unsupervised* machine learning technique in which you train a model to group similar entities into clusters based on their features.

In this exercise, you must separate a dataset consisting of three numeric features (**A**, **B**, and **C**) into clusters.

Let's begin by importing the data.

```
# Load the core tidyverse and make it available in your current R session
library(tidyverse)

# Read the csv file into a tibble
clust_data <- read_csv(file = "https://raw.githubusercontent.com/MicrosoftDocs/ml-basics/master/challeng

# Print the first 10 rows of the data
clust_data %>%
  slice_head(n = 10)
```

```
## # A tibble: 10 x 3
##          A      B       C
##      <dbl>  <dbl>   <dbl>
##  1 -0.0875  0.398  0.0143
##  2 -1.07   -0.546  0.0724
##  3  2.75    2.01   3.08
##  4  3.22    2.21   4.26
##  5 -0.607   0.794 -0.516
##  6 -0.212   0.328 -0.177
##  7  4.02    3.41   3.35
##  8  2.75    1.85   1.37
##  9  0.214  -0.965  0.197
## 10 -0.270  -0.433 -1.01
```

Your challenge is to identify the number of discrete clusters present in the data, and create a clustering model that separates the data into that number of clusters. You should also visualize the clusters to evaluate the level of separation achieved by your model.

Add markdown and code cells as required to create your solution.

Use PCA to create a 2D version of the features for visualization

`Principal Component Analysis` (PCA) is a dimension reduction method that aims at reducing the feature space, such that, most of the information or variability in the data set can be explained using fewer uncorrelated features. Let's see this in action by creating a specification of a `recipe` that will estimate the *principal components* based on our three variables. We'll ater `prep` and`bake` the recipe to apply the computations.

Question 1.

For this question, create a specification of a `recipe` that will: - Normalize all predictors. - Convert all predictors into **two** principal components.

```
BEGIN QUESTION
name: Question 1
manual: false
```

```r
set.seed(2056)
# Load the core tidymodels packages and make them available in your current R session
library(tidymodels)

# Specify a recipe
pca_rec <- recipe(~ ., data = clust_data) %>%
  # Recipe step for normalizing predictors
  step_normalize(all_predictors()) %>%
  # Recipe step for converting all predictors into 2 pcs
  step_pca(all_predictors(), num_comp = 2, id = "pca")

# Print out recipe
pca_rec
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##  predictor          3
##
## Operations:
##
## Centering and scaling for all_predictors()
## No PCA components were extracted.
```

```r
. = " # BEGIN TEST CONFIG

success_message: Great start! You have successfully created a recipe specification that will: Normalize
failure_message: Almost there! Let's take a look at this again. Ensure your recipe specifies that all pr
" # END TEST CONFIG

suppressPackageStartupMessages({
  library(testthat)
  library(ottr)

})

## Test ##
# test_that('data dimensions correct', {
#   expect_output(invisible(print(pca_rec)), "Recipe\n\nInputs:\n\n\nOperations:\n\nCentering and scali
#
# })
```

Well done! Compared to supervised learning techniques, we have no `outcome` variable in this recipe.

Now, time to prep and bake the recipe!

Question 2.

In this section: - Prep the recipe you created above. Prepping a recipe estimates the required quantities and statistics required by PCA. - Bake the prepped recipe. This takes the trained (prepped) recipe and applies its operations to a data set to create a design matrix.

```
BEGIN QUESTION
name: Question 2
manual: false
```

```
# Estimate the recipe
pca_estimates <- prep(pca_rec)

# Return preprocessed data using bake
features_2d <- pca_estimates %>%
  bake(new_data = NULL)

# Print baked data set
features_2d %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 2
##      PC1    PC2
##    <dbl>  <dbl>
## 1 -1.93 -0.239
## 2 -2.80 -0.452
## 3  1.52  0.270
## 4  2.37  0.261
## 5 -2.23 -0.649
```

```
. = " # BEGIN TEST CONFIG

success_message: Fantatsic! You have successfully prepped a recipe and baked it to obtain the fitted PC
failure_message: Almost there! Let's take a look at this again. Ensure you call `prep()` which estimates
" # END TEST CONFIG


## Test ##
# test_that('prep and bake results are correct', {
#   expect_output(invisible(print(pca_estimates)), "Recipe\n\nInputs:\n\n\nTraining data contained 1000
#   expect_equal(dim(features_2d), c(1000, 2))
#
#
# })
```

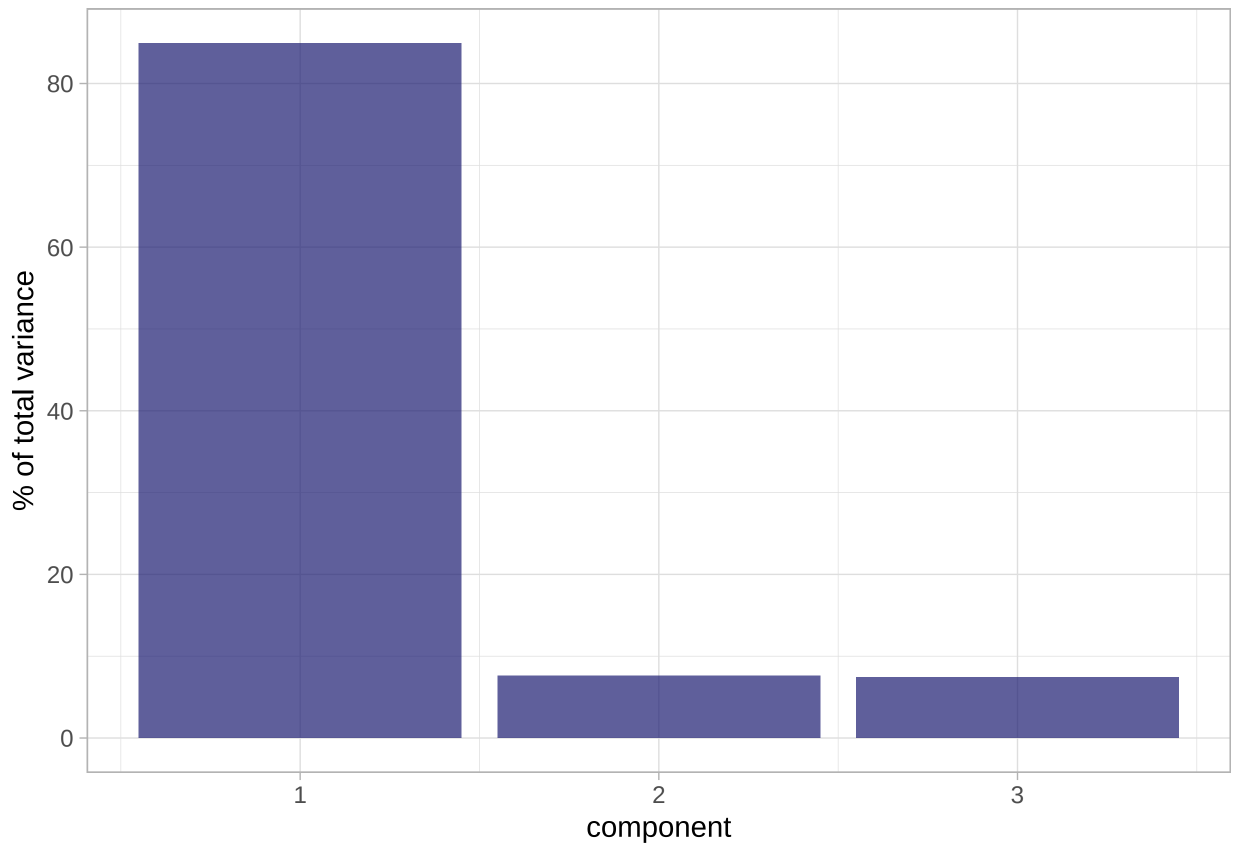
Good job! After prep and bake, you obtain two components which capture the maximum amount of information (i.e. variance) in the original variables.

From the output of our prepped recipe `pca_estimates`, we can examine how much variance each component accounts for:

```r
# Examine how much variance each PC accounts for
pca_estimates %>%
  tidy(id = "pca", type = "variance") %>%
  filter(str_detect(terms, "percent"))
```

```
## # A tibble: 6 x 4
##   terms                        value component id
##   <chr>                        <dbl>     <int> <chr>
## 1 percent variance             84.9          1 pca
## 2 percent variance              7.63         2 pca
## 3 percent variance              7.44         3 pca
## 4 cumulative percent variance  84.9          1 pca
## 5 cumulative percent variance  92.6          2 pca
## 6 cumulative percent variance 100            3 pca
```

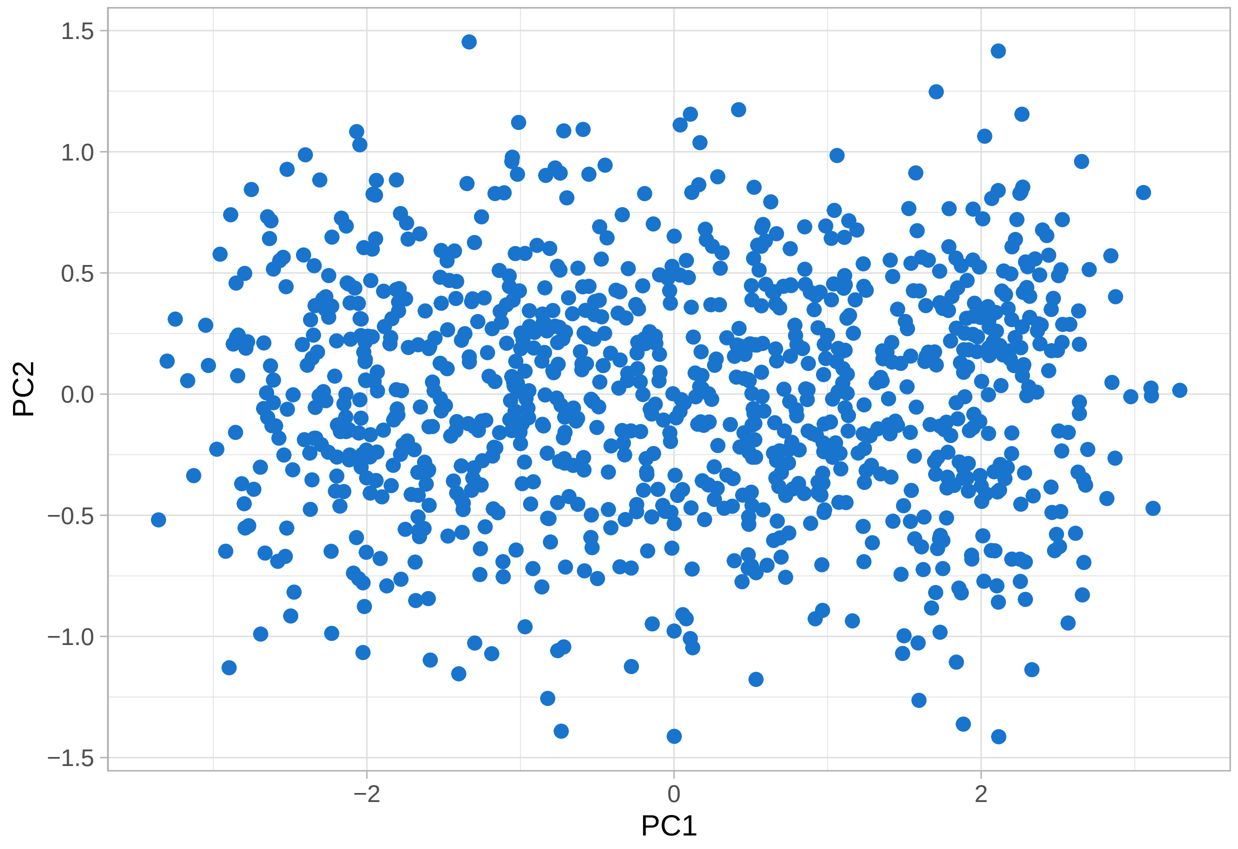
```r
theme_set(theme_light())
# Plot how much variance each PC accounts for
pca_estimates %>%
  tidy(id = "pca", type = "variance") %>%
  filter(terms == "percent variance") %>%
  ggplot(mapping = aes(x = component, y = value)) +
  geom_col(fill = "midnightblue", alpha = 0.7) +
  ylab("% of total variance")
```



This output tibbles and plots shows how well each principal component is explaining the original six variables. For example, the first principal component (PC1) explains about `84.93%` of the variance of the three variables. The second principal component explains an additional 7.63%, giving a cumulative percent variance of 92.56%. This is certainly better. It means that the first two variables seem to have some power in summarizing the original three variables.

Now that we have the data points translated to two dimensions PC1 and PC2, we can visualize them in a plot:

```r
# Visualize PC scores
features_2d %>%
  ggplot(mapping = aes(x = PC1, y = PC2)) +
  geom_point(size = 2, color = "dodgerblue3")
```



Calculate WCSS for multiple cluster numbers to determine the right number of clusters

Now here lies one of the fundamental problems with clustering - without known class labels, how do you know how many clusters to separate your data into?

Question 3.

In this section, we'll try to find the optimal number of clusters our data can be divided into by using our data sample, `clust_data`, to create a series of clustering models with an incrementing number of clusters, and then measure how tightly the data points are grouped within each cluster.

A metric often used to measure this tightness is the *within cluster sum of squares* (WCSS), with lower values meaning that the data points are closer. You can then plot the WCSS for each model.

- Prep and bake `clust_data` to obtain a new tibble where all the predictors are normalized.

- Use the built-in `kmeans()` function to perform clustering by creating a series of 10 clustering models.
- Extract the WCSS, `tot.withinss` for each cluster.

```
BEGIN QUESTION
name: Question 3
manual: false
```

```
# Data for clustering
clust_estimate_data <- recipe(~ ., data = clust_data) %>%
  step_normalize(all_predictors()) %>%
  prep() %>%
  bake(new_data = NULL)

# Print out data
clust_estimate_data %>%
  slice_head(n = 5)
```

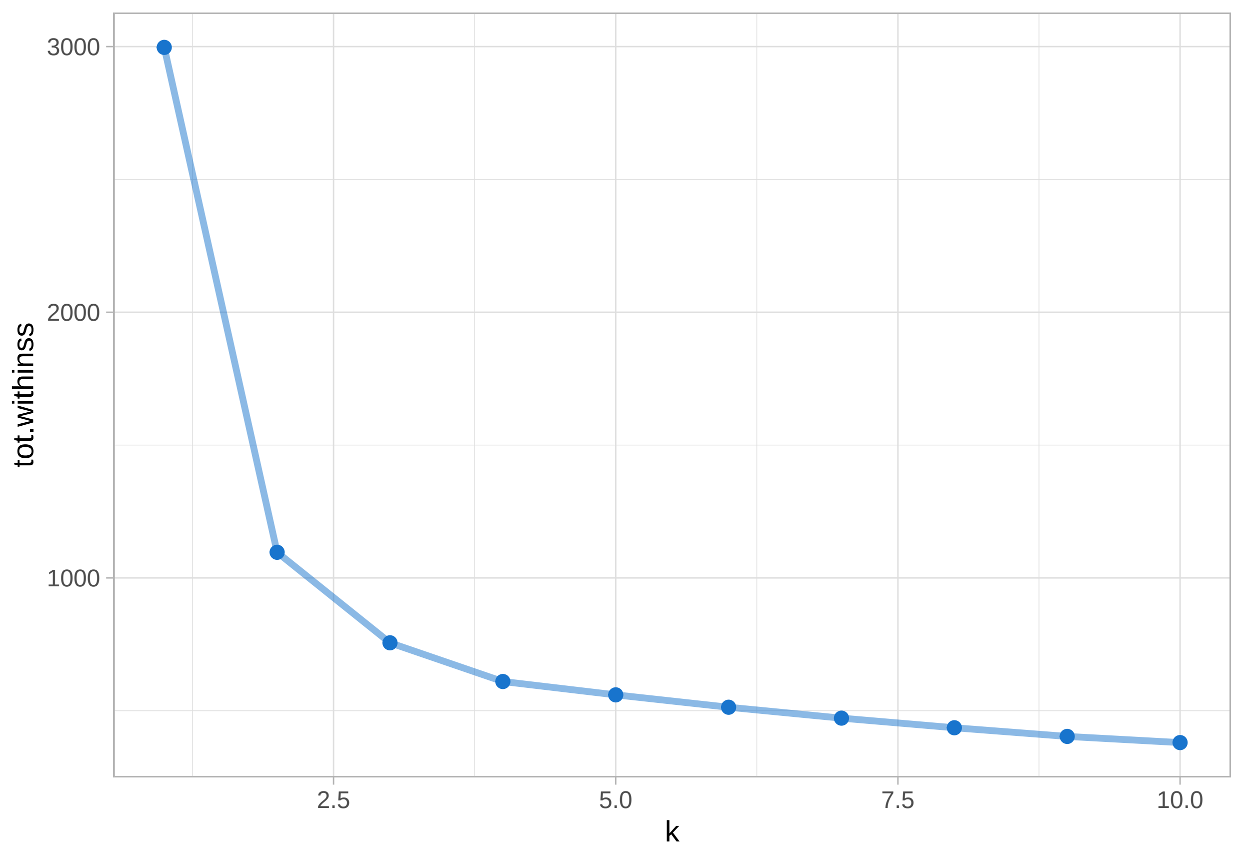
```
## # A tibble: 5 x 3
##        A      B     C
##    <dbl>  <dbl> <dbl>
## 1 -1.28  -0.900 -1.17
## 2 -2.07  -1.66  -1.13
## 3  0.987  0.394  1.25
## 4  1.36   0.555  2.18
## 5 -1.70  -0.583 -1.59
```

Now, let's explore the WCSS of different numbers of clusters.

We'll get to use `map()` from the purrr package to apply functions to each element in list.

> `map()` functions allow you to replace many for loops with code that is both more succinct and easier to read. The best place to learn about the `map()` functions is the iteration chapter in R for data science.
>
> `broom::glance.kmeans()` accepts a model object and returns a tibble with exactly one row of model summaries. The summaries are typically goodness of fit measures, p-values for hypothesis tests on residuals, or model convergence information.

```
set.seed(2056)
# Create 10 models with 1 to 10 clusters
kclusts <- tibble(k = 1:10) %>%
  mutate(
    # Create a series of clustering models with centers k
    model = map(k, ~ kmeans(x = clust_estimate_data, centers = .x, nstart = 20)),
    # Extract the Total WCSS tot.withinss, for each model
    tot.withinss = map_dbl(model, ~ glance(.x)$tot.withinss))



# Plot Total within-cluster sum of squares (tot.withinss) for each model with center k
kclusts %>%
  ggplot(mapping = aes(x = k, y = tot.withinss)) +
  geom_line(size = 1.2, alpha = 0.5, color = "dodgerblue3") +
  geom_point(size = 2, color = "dodgerblue3")
```



```
. = " # BEGIN TEST CONFIG

success_message: Fantatsic! You have successfully created a series of clustering models with an incremen
failure_message: Almost there! Let's take a look at this again. Expected tibble dimensions are [10 3] an
" # END TEST CONFIG


## Test ##
test_that('cluster estimation is looking good', {
  expect_equal(dim(kclusts), c(10, 3))
  expect_true(identical(map_dbl(kclusts$model, ~ glance(.x)$tot.withinss), kclusts$tot.withinss))


})
```

```
## Test passed
```

That went well! What do you think the chart is telling us?

We seek to **minimize** the the total within-cluster sum of squares, by performing K-means clustering. The plot shows a large reduction in WCSS (so greater *tightness*) as the number of clusters increases from one to two, and a further noticable reduction from two to three, then three to four clusters. After that, the reduction is less pronounced, resulting in an `elbow` in the chart at around four clusters. This is a good indication that there are four reasonably well separated clusters of data points.

Use K-Means

Question 4.

In the previous section, we noticed that there is a good indication that there are four reasonably well separated clusters of data points. In this section:

Extract the model that will separate the data into k = 4 clusters. Starting with the clustering results `kclusts`: - create a subset where the models contains `k = 4` clusters. - And then, using `pull()`, extract the clustering model in the `model` column. - And then retrieve the model components, such as cluster assignments, using `pluck()`

```
BEGIN QUESTION
name: Question 4
manual: false
```

```
# Extract the model with k = 4 clusters
final_kmeans <- kclusts %>%
  filter(k == 4) %>%
  pull(model) %>%
  pluck(1)

# Print model components
final_kmeans
```

```
## K-means clustering with 4 clusters of sizes 259, 260, 253, 228
##
## Cluster means:
##            A          B          C
## 1  1.1757290  1.1827466  1.1717490
## 2  0.3454534  0.3627092  0.3634415
## 3 -0.4398796 -0.4734455 -0.4421777
## 4 -1.2414130 -1.2318159 -1.2548544
##
## Clustering vector:
##    [1] 4 4 1 1 4 4 1 2 4 4 1 3 3 2 2 2 1 3 2 2 4 2 4 4 2 3 1 4 3 2 3 4 1 2 2 4 2
##   [38] 2 4 2 2 2 1 3 1 4 4 1 1 2 4 3 1 1 1 1 2 4 1 4 4 4 4 4 2 2 3 2 3 1 3 2 2 1
##   [75] 3 2 2 2 4 3 2 3 1 3 2 1 3 2 2 3 3 3 1 2 3 2 1 2 1 3 2 3 1 3 2 3 1 2 2 1 2
##  [112] 3 4 4 3 1 3 1 4 3 2 3 1 1 4 2 3 2 1 4 3 3 3 2 3 3 2 4 2 3 4 3 3 4 1 1 2 1
##  [149] 2 2 1 4 4 2 4 3 4 2 2 2 3 4 1 1 2 3 3 4 4 1 4 3 2 1 2 3 3 2 2 4 2 4 3 3 2
##  [186] 1 4 1 1 3 3 4 3 2 1 3 2 2 4 1 3 1 1 1 4 4 1 2 1 4 1 1 3 4 1 2 2 1 4 1 2 4
##  [223] 3 1 3 2 3 4 3 4 2 4 3 2 4 1 1 4 1 3 1 3 1 2 2 3 1 4 4 2 1 2 3 3 1 1 1 4 1
##  [260] 4 2 1 1 2 4 2 3 3 3 2 1 4 2 3 4 3 1 2 2 4 1 1 3 4 4 1 1 3 1 4 4 3 4 1 1 1
##  [297] 2 2 3 4 1 2 1 3 4 2 4 2 3 4 3 4 4 3 3 4 4 4 3 3 4 2 4 2 4 1 4 4 2 2 2 3 4
##  [334] 1 1 2 2 3 3 2 1 3 4 2 1 2 2 4 2 3 4 2 1 1 1 1 4 2 1 3 3 3 4 4 2 1 3 1 2 1
##  [371] 2 1 3 4 1 4 3 1 4 2 3 3 2 2 3 2 1 2 2 4 2 1 3 1 3 1 2 2 3 1 4 1 3 4 1 4 2
##  [408] 3 4 3 1 2 3 1 3 1 1 3 1 4 4 4 4 1 2 3 2 4 2 2 4 4 3 2 3 1 4 3 1 4 1 1 4 1
##  [445] 2 3 1 1 4 2 1 2 2 4 1 2 1 3 2 3 1 3 2 4 4 2 1 2 2 3 2 2 1 3 4 4 2 3 4 4 4
##  [482] 3 1 3 4 1 4 1 4 3 3 1 2 4 2 1 1 1 1 4 1 2 2 3 4 3 3 4 1 2 3 3 4 1 2 1 3 3
##  [519] 1 4 1 1 4 4 2 2 2 2 3 3 4 4 4 3 4 3 2 3 2 2 3 2 2 1 1 1 2 4 3 4 2 1 3 2 2
##  [556] 1 2 2 3 1 1 4 1 3 1 4 2 1 3 1 3 2 3 1 2 2 2 3 4 1 3 4 2 3 4 3 1 4 2 3 1 3
##  [593] 4 2 2 4 2 1 3 4 4 2 4 4 3 4 1 2 2 3 3 4 1 3 3 3 3 4 3 2 4 4 2 3 2 2 1 4 4
##  [630] 1 4 3 2 3 3 2 1 2 3 3 1 1 1 2 1 1 3 3 2 1 3 1 2 3 1 1 1 1 4 2 2 4 1 4 4 3
##  [667] 2 4 4 2 3 3 2 4 1 3 3 1 1 1 1 1 1 2 1 1 3 4 1 1 4 2 2 4 2 3 1 3 4 4 2 4 4
##  [704] 2 1 1 2 1 4 1 4 3 3 3 4 2 3 4 3 1 4 3 4 2 4 3 3 3 4 1 2 2 3 2 3 3 4 2 2 1
##  [741] 3 3 2 3 4 3 4 4 4 4 2 3 4 3 1 3 2 2 2 3 4 2 4 2 2 1 2 2 3 3 4 2 4 1 3 1 1
##  [778] 4 1 2 4 3 3 3 2 1 1 1 2 4 3 1 4 1 4 1 4 4 1 1 4 2 1 3 2 1 2 1 1 3 3 4 1 3
##  [815] 3 2 4 1 2 2 2 1 1 1 1 3 4 4 1 1 1 4 3 3 1 3 1 2 3 3 3 1 1 4 2 3 1 1 2 2 2
##  [852] 2 2 3 2 1 3 3 3 3 1 2 3 2 1 2 2 1 2 2 3 1 2 4 3 2 1 3 3 1 4 1 3 1 3 1 1 3
##  [889] 1 3 2 4 2 2 3 1 2 3 4 4 2 3 2 4 4 2 3 3 4 4 1 1 4 4 1 4 2 3 3 4 2 2 3 1 2
##  [926] 1 2 1 1 4 1 1 1 2 4 1 3 4 1 3 3 4 3 3 2 4 4 1 1 4 3 1 3 3 2 2 4 1 2 2 4 4
##  [963] 2 3 2 4 2 3 2 3 4 3 1 2 2 3 1 1 4 3 3 2 2 1 2 1 4 3 3 1 3 3 3 3 1 2 1 2 4
## [1000] 2
##
## Within cluster sum of squares by cluster:
## [1] 172.2036 147.1231 149.6161 141.0509
##  (between_SS / total_SS =  79.6 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

```
. = " # BEGIN TEST CONFIG

success_message: Good job! You have successfully extracted the model that will separate the data into k
failure_message: Almost there! Let's take a look at this again. Ensure you filter the data set to obtai
" # END TEST CONFIG


## Test ##
test_that('fitted cluster method is correct', {
  expect_equal(length(final_kmeans$cluster), nrow(clust_data))
  expect_equal(range(final_kmeans$cluster), c(1, 4))


})
```

```
## Test passed
```

What can you make out of the kmeans object, `final_kmeans`? One of the available components is the `Clustering vector`: A vector of integers (from 1:k) indicating the cluster to which each observation in our data is allocated. Be sure to explore the other available components.

Plot the clustered points

Great! Let's go ahead and visualize the clusters obtained. Care for some interactivity using `plotly`?

Question 5.

In this section:

- Add the cluster assignments generated by the model to the 2D version (2 Principal Components) of our data set, i.e `features_2d`. Hint: `augment`accepts a model object and a dataset and adds information about each observation in the dataset.
- Visualize the cluster assignments using a scatter plot. Assign the `shape` and `color` aesthetics to the cluster number assigned to each data point.

```
BEGIN QUESTION
name: Question 5
manual: false
```

```r
# Load plotting libraries
library(plotly)
library(paletteer)


# Augment featured_2d with information from the clustering model
results <-  augment(final_kmeans, features_2d)

# Print some rows of the results
results %>%
  slice_head(n = 10)
```

```
## # A tibble: 10 x 3
##       PC1    PC2 .cluster
##     <dbl>  <dbl> <fct>
##  1 -1.93  -0.239 4
##  2 -2.80  -0.452 4
##  3  1.52   0.270 1
##  4  2.37   0.261 1
##  5 -2.23  -0.649 4
##  6 -2.11  -0.251 4
##  7  2.88   0.402 1
##  8  0.667  0.662 2
##  9 -2.34   0.530 4
## 10 -2.87   0.207 4
```

Voila! There goes the two Principal Components (PC1, PC2) which capture the maximum amount of information (i.e. variance) in the original variables along with their corresponding cluster assignment (.cluster).

Now, let's see this in an interactive plot!

```r
# Plot cluster assignments
cluster_plot <- results %>%
  ggplot(mapping = aes(x = PC1, y = PC2)) +
  geom_point(aes(shape = .cluster, color = .cluster), size = 2, alpha = 0.8) +
  paletteer::scale_color_paletteer_d("ggsci::category10_d3")

# Make plot interactive
ggplotly(cluster_plot)
```

```r
. = " # BEGIN TEST CONFIG

success_message: Good job! You have successfully visualized the cluster assignmnets.
failure_message: Almost there! Let's take a look at this again. Ensure you have assigned the `shape` and
" # END TEST CONFIG



## Test ##
test_that('Cluster plot is looking good', {
  expect_output(print(cluster_plot$layers[[1]]), "mapping: shape = ~.cluster, colour = ~.cluster \ngeom_



})
```

```
## Test passed
```

```
. = " # BEGIN TEST CONFIG

success_message: Good job! You have successfully augmented the Principal Components with corresponding 
failure_message: Almost there! Let's take a look at this again. Ensure you have correctly used augment(
" # END TEST CONFIG


## Test ##
test_that('Principal Components are augmented with clustering results', {
  expect_equal(results %>% select(!contains("PC")) %>% pull() %>% as.numeric() %>% range(), c(1, 4))
  expect_equal(dim(results), c(1000, 3))


})
```

```
## Test passed
```

Fantastic! Hopefully, the data has been separated into four distinct clusters.

K-Means is a commonly used clustering algorithm that separates a dataset into K clusters of equal variance such that observations within the same cluster are as similar as possible (i.e., high intra-class similarity), whereas observations from different clusters are as dissimilar as possible (i.e., low inter-class similarity).

The first step in K-Means clustering is the data scientist specifying the number of clusters K to partition the observations into. Hierarchical clustering is an alternative approach which does not require the number of clusters to be defined in advance. Let's jump right into it.

Try agglomerative clustering

Hierarchical clustering creates clusters by either a *divisive* method or *agglomerative* method. The `divisive` method is a `top down` approach starting with the entire dataset and then finding partitions in a stepwise manner. `Agglomerative clustering` is a `bottom up` approach in which each observation is initially considered as a single element cluster and at each step of the algorithm, two clusters that are most similar are combined into a bigger cluster until all points are members of one single big cluster.

Question 6.

In this section we'll cluster the data using an agglomerative clustering algorithm. There are many functions available in R for hierarchical clustering.

The `hclust()` function is one way to perform hierarchical clustering. It requires a distance matrix structure and a specification of the agglomeration method to be used.

- Use the `dist` function to compute the distance matrix using the "euclidean" method to compute the distances between the rows of the data.

- Use `hclust` to perform Hierarchical clustering with the "ward.D2" (Ward linkage) agglomeration method.

Please see the corresponding Learn module for more information on the above.

```
BEGIN QUESTION
name: Question 6
manual: false
```

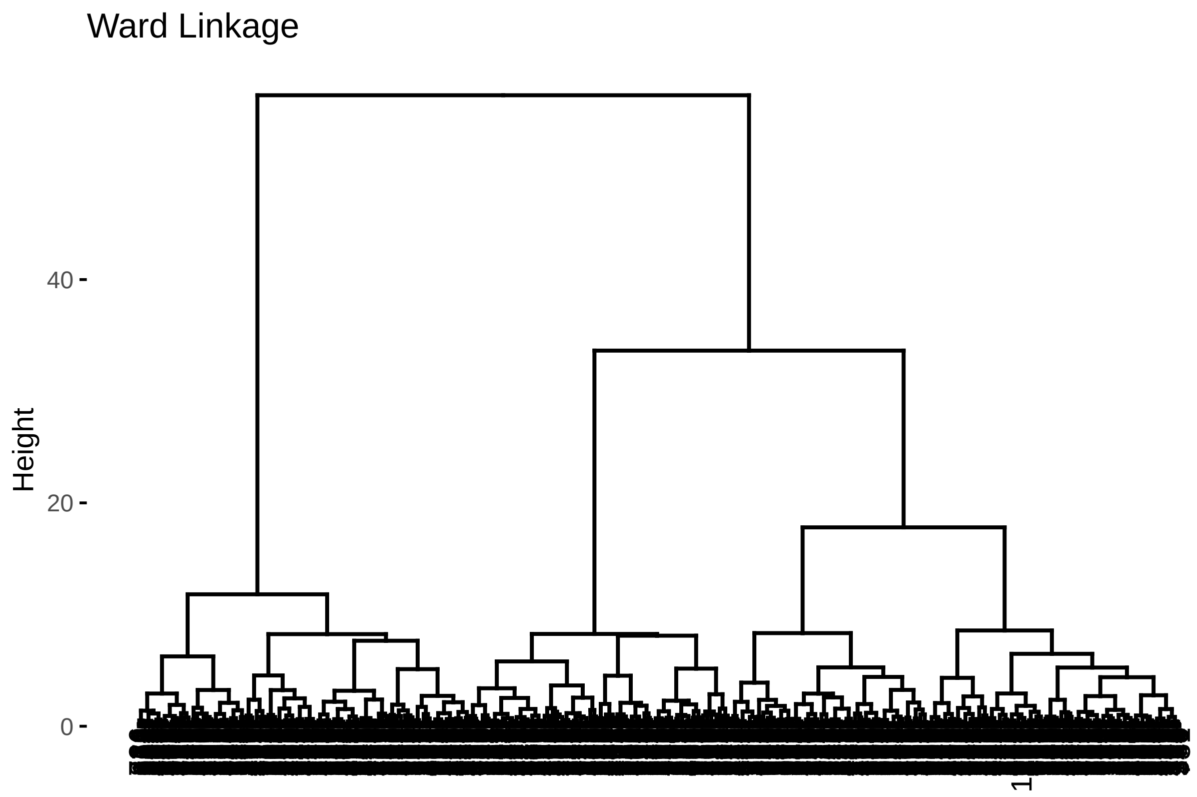
```r
# For reproducibility
set.seed(2056)

# Distance between observations matrix
d <- dist(x = clust_estimate_data, method = "euclidean")

# Hierarchical clustering using Ward Linkage
hclust_ward <- hclust(d, method = "ward.D2")
```

```r
. = " # BEGIN TEST CONFIG

success_message: Fantastic! You have successsfully performed hierarchical clustering using the Ward lin
failure_message: Almost there! Let's take a look at this again. Ensure you have used the Ward linkage m
" # END TEST CONFIG


## Test ##
test_that('hierarchical clustering was computed', {
  expect_equal(hclust_ward$dist.method, "euclidean")
  expect_equal(hclust_ward$method, "ward.D2")


})
```

```
## Test passed
```

The factoextra provides functions (`fviz_dend()`) to visualize hierarchical clustering. Let's visualize the dendrogram representation of the clusters from the ward linkage method.

```r
library(factoextra)

# Visualize cluster separations
fviz_dend(hclust_ward, main = "Ward Linkage")
```

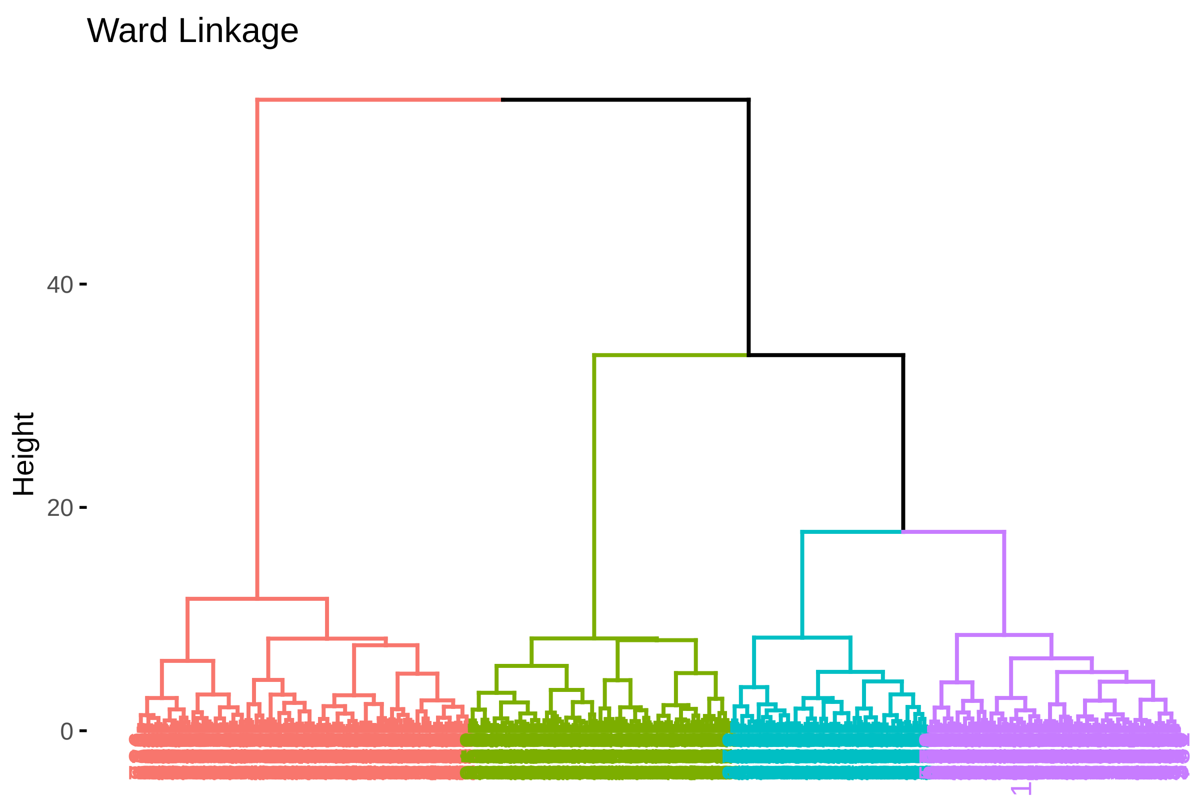


Although hierarchical clustering does not require one to pre-specify the number of clusters, one still needs to specify the number of clusters to extract.

We found out, using the `WCSS` method, that the optimal number of clusters was 4. Let's color our dendrogram according to k = 4 and observe how observations will be grouped.

```
# Visualize clustering structure for 4 groups
fviz_dend(hclust_ward, k = 4, main = "Ward Linkage")
```

```
## Warning: 'guides(<scale> = FALSE)' is deprecated. Please use 'guides(<scale> =
## "none")' instead.
```



Perfect! We can now go ahead and `cut` the hierarchical clustering model into four clusters and extract the cluster labels for each observation associated with a given cut. This is done using `cutree()`

Question 7.

In this section, we are going to `cut` the hierarchical clustering model into four clusters and extract the cluster labels for each observation.

Starting with the Principal Components version of the data:

- Extract k = 4 groups of data obtained from the Hierarchical clustering process. Assign this to a column name `cluster_id`
- Also, encode the `cluster_id` column as a factor.

```
BEGIN QUESTION
name: Question 7
manual: false
```

```
set.seed(2056)
# Extract cluster labels obtained from the Hierarchical clustering process
results_hclust <- features_2d %>%
  mutate(cluster_id = factor(cutree(hclust_ward, k = 4)))


# Print some rows of the results
results_hclust %>%
  slice_head(n = 5)
```

```
## # A tibble: 5 x 3
##     PC1    PC2 cluster_id
##   <dbl>  <dbl> <fct>
## 1 -1.93 -0.239 1
## 2 -2.80 -0.452 1
## 3  1.52  0.270 2
## 4  2.37  0.261 2
## 5 -2.23 -0.649 1
```

```
. = " # BEGIN TEST CONFIG

success_message: Fantastic! You have successsfully augmented the Principal Components with cluster label
failure_message: Almost there! Let's take a look at this again. Ensure you have extracted 4 groups of da
" # END TEST CONFIG


## Test ##
test_that('hierarchical clustering was computed', {
  expect_equal(dim(results_hclust), c(1000, 3))
  expect_equal(class(results_hclust$cluster_id), "factor")
  expect_equal(range(as.numeric(results_hclust$cluster_id)), c(1, 4))


})
```

```
## Test passed
```

Awesome! Now, let's inspect the cluster assignmnents visually.

View the agglomerative cluster assignments

```
# Plot h-cluster assignmnet on the PC data
hclust_spc_plot <- results_hclust %>%
  ggplot(mapping = aes(x = PC1, y = PC2)) +
  geom_point(aes(shape = cluster_id, color = cluster_id), size = 2, alpha = 0.8) +
  paletteer::scale_color_paletteer_d("ggsci::category10_d3")

# Make plot interactive
ggplotly(hclust_spc_plot)
```

Neat! How does this compare to the K-Means results we visualized earlier?

We can see that the observations were grouped quite similarly by the two clustering algorithms, **K-Means Clustering** and **Hierarchical Clustering**.

We'll wrap it at that! In this challenge, you created unsupervised machine learning models, i.e K-Means and Hierarchical clustering models, to help you group unlabeled data observations into clusters.

Congratulations on this amazing feat!

Happy Learning,

Eric, Gold Microsoft Learn Student Ambassador.