Methods for Unsupervised Clustering

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Co-Director, Center for Biomedical Informatics and Health AI
Rutgers University – New Jersey Medical School

2025-11-04

Supervised vs. Unsupervised Machine Learning

Machine learning algorithms are generally classified into two categories. In **Supervised** machine learning we use the outcomes in a training set to **supervise** the creation of our prediction algorithm.

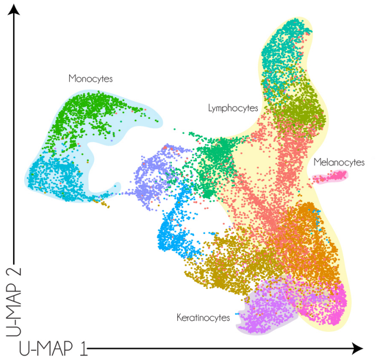
In **unsupervised** machine we do not necessarily know the outcomes and instead are interested in discovering groups. These algorithms are also referred to as **clustering** algorithms since predictors are used to define **clusters**.

Supervised vs. Unsupervised Machine Learning

Sometimes clustering is not be very useful. For example, if we are simply given the heights we may not be able to discover two groups, males and females.

Unsupervised Machine Learning

However, there are applications in which unsupervised learning can be a powerful technique, such as an exploratory tool:



Unsupervised Machine Learning

There are many algorithms for unsupervised learning. We have already learned about **PCA** and **UMAP** for dimension reduction. Here we introduce two methods for clustering: **hierarchical clustering** and **k-means**.

Unsupervised Machine Learning and clustering

A first step in any clustering algorithm is defining a distance between observations or groups of observations.

Hierarchical clustering starts by defining each observation as a separate group, and distances are calculated between every group (distance matrix). Then the two closest groups are merged into a single group, and this new group (two observations) is represented by its centroid. Distances between this new group and the rest are calculated, and then the next two closest groups are merged. This process is repeated until there is just one group.

Hierarchical clustering

Consider the ratings of 50 movies from 139 different critics:

```r
library(dslabs); data("movielens")
top <- movielens %>% group_by(movieId) %>%
  summarize(n=n(), title = first(title)) %>%
  top_n(50, n) %>% pull(movieId)

x <- movielens %>%filter(movieId %in% top) %>%
  group_by(userId) %>% filter(n() >= 25) %>%
  ungroup() %>% select(title, userId, rating) %>%
  spread(userId, rating)

row_names <- str_remove(x$title, ": Episode") %>% str_trunc(20)
x <- x[,-1] %>% as.matrix()
x <- sweep(x, 2, colMeans(x, na.rm = TRUE))
x <- sweep(x, 1, rowMeans(x, na.rm = TRUE))
rownames(x) <- row_names
```

Hierarchical clustering

We want to use these data to find out if there are clusters of movies based on the ratings from 139 movie raters. A first step is to find the distance between each pair of movies using the `dist` function:

```r
d <- dist(x)
```

Hierarchical clustering

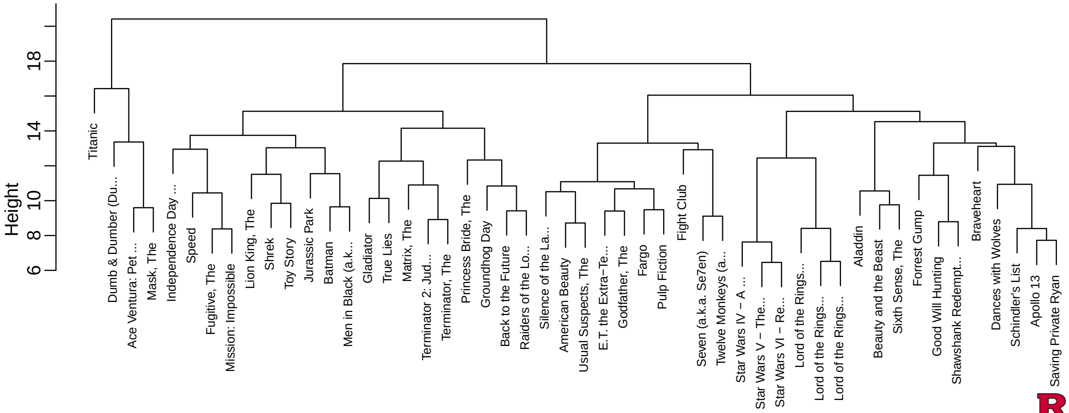
With the distance between each pair of movies computed, we need an algorithm to define groups from these. The `hclust` function implements this algorithm and it takes a distance as input.

```r
h <- hclust(d)
h
```

```
##
## Call:
## hclust(d = d)
##
## Cluster method   : complete
## Distance         : euclidean
## Number of objects: 50
```

Hierarchical clustering

We can see the resulting groups using a **dendrogram**.

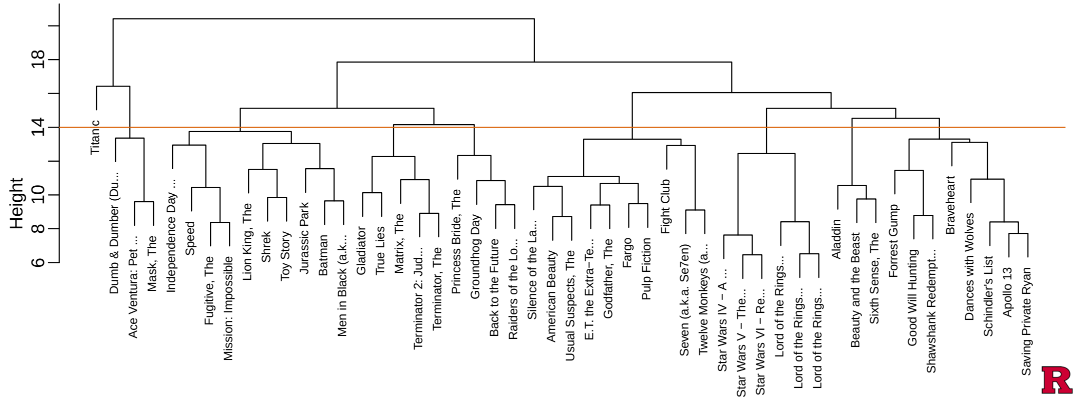


Hierarchical clustering

This graph gives us an approximation between the distance between any two movies. To find this distance we find the first location, from top to bottom, where these movies split into two different groups. The height of this location is the distance between these two groups. So, for example, the distance between the three *Star Wars* movies is 8 or less, while the distance between *Raiders of the Lost of Ark* and *Silence of the Lambs* is about 17.

Hierarchical clustering

To generate actual groups: 1) decide on a maximum distance to be in the same group or 2) decide on the number of groups. For example:



Hierarchical clustering

The function `cutree` can be applied to the output of `hclust` to perform either of these two operations and generate groups.

```
# Maximum Distance
groups <- cutree(h, h = 14)
table(groups)
```

```
## groups
##  1  2  3  4  5  6  7  8  9
##  3  3 10  8  4 10  5  6  1
```

Hierarchical clustering

Or we can do fewer groups:

```
#Number of groups
groups <- cutree(h, k = 10)
table(groups)
```

```
## groups
##  1  2  3  4  5  6  7  8  9 10
##  3  3 10  8  4  6  4  5  6  1
```

Hierarchical clustering

The clustering provides some insights, e.g., Group 4 appears to be blockbusters:

```
names(groups)[groups==4]
```

```
## [1] "Apollo 13"          "Braveheart"          "Dances with Wolves"
## [4] "Forrest Gump"       "Good Will Hunting"   "Saving Private Ryan"
## [7] "Schindler's List"   "Shawshank Redempt..."
```

And group 9 appears to be fantasy/nerd movies:

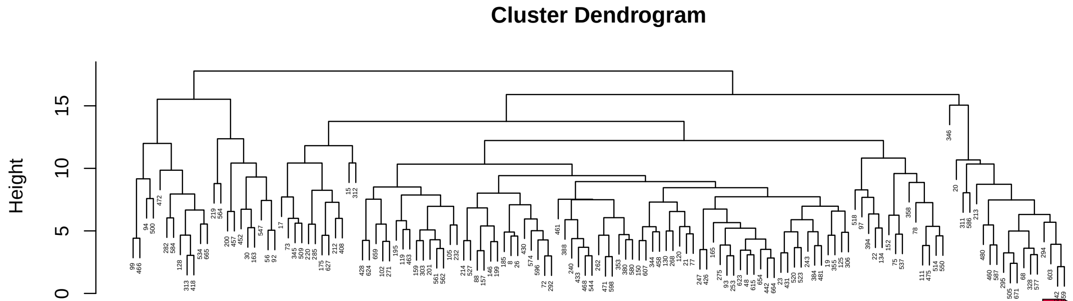
```
names(groups)[groups==9]
```

```
## [1] "Lord of the Rings..." "Lord of the Rings..." "Lord of the Rings..."
## [4] "Star Wars IV - A ..." "Star Wars V - The..." "Star Wars VI - Re..."
```

Hierarchical clustering

We can also explore the data to see if there are clusters of movie raters.

```r
h_2 <- dist(t(x)) %>% hclust()
plot(h_2, cex = 0.35)
```



K-Means

We can also use the k-means algorithm. Here, we have to pre-define k, the number of clusters we want to define.

The k-means algorithm is iterative. The first step is to define k centers. Then each observation is assigned to the cluster with the closest center to that observation. In a second step the centers are redefined using the observation in each cluster: the column means are used to define a **centroid**. We repeat these two steps until the centers converge.

K-Means Clustering: The Objective Function

The goal of the k-means algorithm is to minimize the **Within-Cluster Sum of Squares (WCSS)**, also known as **Inertia** or the **quantization error**.

The WCSS is the sum of the squared Euclidean distances between each observation and its assigned cluster centroid.

$$\min_{S} \sum_{k=1}^{K} \sum_{\mathbf{x}_i \in S_k} ||\mathbf{x}_i - \boldsymbol{\mu}_k||^2$$

K-Means Clustering: The Objective Function

Where:

- K is the number of clusters.
- S_k is the set of observations in cluster k.
- $\mathbf{x}_i$ is an observation vector.
- $\boldsymbol{\mu}_k$ is the centroid (mean) of cluster S_k.

The algorithm iteratively seeks to reduce this value.

K-Means Clustering: Convergence

The standard k-means algorithm (Lloyd's algorithm) is an **iterative refinement technique** that alternates between two steps:

1. **Assignment (Expectation) Step:** Assign each observation to the cluster whose centroid is the closest. This minimizes $||\mathbf{x}_i - \boldsymbol{\mu}_k||^2$ for a fixed $\boldsymbol{\mu}_k$.
2. **Update (Maximization) Step:** Recalculate the centroids $\boldsymbol{\mu}_k$ as the mean of all observations assigned to cluster k. This minimizes the WCSS for fixed cluster assignments S_k.

K-Means Clustering: Convergence

Each step strictly **decreases** the WCSS until the assignments or centroids no longer change. This property guarantees that the algorithm will **always converge** in a finite number of iterations.

K-Means Clustering: Local vs. Global Optimum

While the algorithm is guaranteed to converge, it is **not guaranteed to find the global minimum** of the WCSS objective function.

- The final solution is a **local minimum**, meaning any small change in the cluster assignments or centroids would increase the WCSS.
- The result is highly dependent on the **initial choice of the** K **cluster centers**.

K-Means Clustering: Local vs. Global Optimum

- To address multiple convergence, we commonly run the algorithm multiple times (e.g., using `nstart` in R) with different random initializations and select the solution with the lowest final WCSS.
- Methods like **k-means++** (not used in base R's `kmeans` function) are designed to select better initial centroids to increase the chance of finding a good local minimum.

K-Means

The `kmeans` function included in R-base does not handle NAs. For illustrative purposes we will fill out the NAs with 0s. In general, the choice of how to fill in missing data, or if one should do it at all, should be made with care.

```
x_0 <- x; x_0[is.na(x_0)] <- 0
set.seed(0)
k <- kmeans(x_0, centers = 10)
```

k-means

The cluster assignments are in the `cluster` component:

```r
groups <- k$cluster
table(groups)
```

```
## groups
##  1  2  3  4  5  6  7  8  9 10
##  1  9  3  3  4  3  3  5  6 13
```

k-means

This yields some interesting groups:

```r
names(groups)[groups==4]
```

```
## [1] "Ace Ventura: Pet ..." "Dumb & Dumber (Du..." "Mask, The"
```

```r
names(groups)[groups==6]
```

```
## [1] "Lord of the Rings..." "Lord of the Rings..." "Lord of the Rings..."
```

```r
names(groups)[groups==7]
```

```
## [1] "Star Wars IV - A ..." "Star Wars V - The..." "Star Wars VI - Re..."
```

```r
names(groups)[groups==9]
```

```
## [1] "Braveheart"           "Dances with Wolves"   "Godfather, The"
## [4] "Good Will Hunting"    "Schindler's List"     "Shawshank Redempt..."
```

```r
names(groups)[groups==10]
```

```
##  [1] "Aladdin"              "Apollo 13"            "Beauty and the Beast"
##  [4] "E.T. the Extra-Te..." "Forrest Gump"         "Gladiator"
```

k-means

Note that because the first center is chosen at random, the final clusters are random. We impose some stability by repeating the entire function several times and averaging the results. The number of random starting values to use can be assigned through the `nstart` argument.

```r
k <- kmeans(x_0, centers = 10, nstart = 25)
```

Other Unsupervised Clustering Methods

While hierarchical clustering and k-means are fundamental, many other powerful clustering algorithms exist.

Algorithm	Method Class	Key Features
Mixture Modeling (GMM)	Probabilistic/Model-Based	Assumes data is generated from a mixture of K probability distributions (e.g., Gaussian). Provides **soft clustering**. Uses the **Expectation-Maximization (EM) algorithm**.
DBSCAN	Density-Based	Finds clusters of arbitrary shape based on **density**. Can identify **outliers** (noise). Does not require pre-specifying K.
Louvain Method	Graph/Community Detection	Highly efficient for finding **communities** in large **networks/graphs** by optimizing **modularity**.
Affinity Propagation	Affinity/Message Passing	Identifies "exemplars" as cluster centers. Does **not** require pre-specifying K.

Community Detection: The Louvain Method

The Louvain method is an efficient algorithm for detecting **communities** (or clusters) in large **networks** or **graphs**. It's widely used in computational biology (e.g., single-cell analysis) and social network analysis.

Instead of minimizing WCSS like k-means, Louvain aims to maximize a metric called **modularity** ($\mathcal{Q}$).

$$\mathcal{Q} = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

Community Detection: The Louvain Method

Where:

- A_{ij} is the weight of the edge between nodes i and j.
- m is the total weight of all edges in the network.
- k_i and k_j are the sum of weights of edges attached to nodes i and j (their degrees).
- $\delta(c_i, c_j)$ is 1 if nodes i and j belong to the same community, and 0 otherwise.

Modularity measures the strength of the division of a network into modules (or communities). A high modularity value indicates that communities have **dense internal connections** but **sparse connections** between them.

Louvain Method: The Algorithm

The Louvain method is a **greedy optimization** algorithm that proceeds in two main phases, which are repeated iteratively until no further increase in modularity is possible:

Louvain Method: The Algorithm

Phase 1: Modularity Optimization

1. Start with every node in the graph as its own **community**.
2. For each node i, the algorithm considers moving it to all of its neighbors' communities.
3. It calculates the **gain in modularity** ($\Delta\mathcal{Q}$) for each move.
4. Node i is moved to the community that yields the largest positive $\Delta\mathcal{Q}$. If no move increases $\mathcal{Q}$, the node stays put.

Louvain Method: The Algorithm

Phase 2: Community Aggregation

1. A new **coarse-grained** network is built:
    - Each community found in Phase 1 becomes a **single node** (a *super-node*).
    - The weight of the edges between these super-nodes is the sum of the weights of the edges between the original communities.
2. The algorithm then re-applies Phase 1 to this new, smaller network.

Benefits: The method is extremely **fast** and effectively finds high-modularity partitions, making it suitable for graphs with **millions of nodes**.

Heatmaps

A powerful visualization tool for discovering clusters or patterns in your data is the heatmap. The idea is simple: plot an image of your data matrix with colors used as the visual cue and both the columns and rows ordered according to the results of a clustering algorithm. We will demonstrate this with the `tissue_gene_expression` dataset. We will scale the rows of the gene expression matrix.

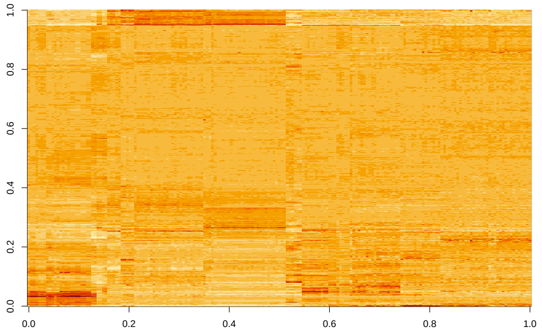
The first step is compute:

```r
data("tissue_gene_expression")
x <- sweep(tissue_gene_expression$x, 2,
           colMeans(tissue_gene_expression$x))
h_1 <- hclust(dist(x))
h_2 <- hclust(dist(t(x)))
```

Heatmaps

Now we can use the results of this clustering to order the rows and columns.

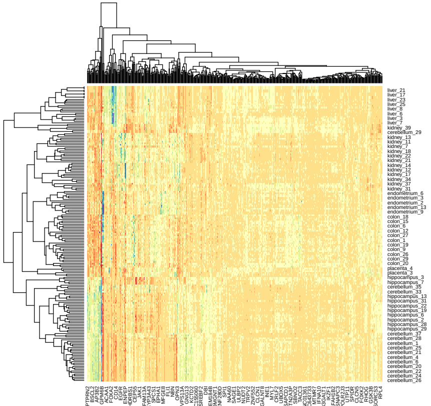
```r
image(x[h_1$order, h_2$order])
```



Heatmaps

But there is `heatmap` function that does it for us:

```r
heatmap(x, col = RColorBrewer::brewer.pal(11, "Spectral"))
```



Filtering features

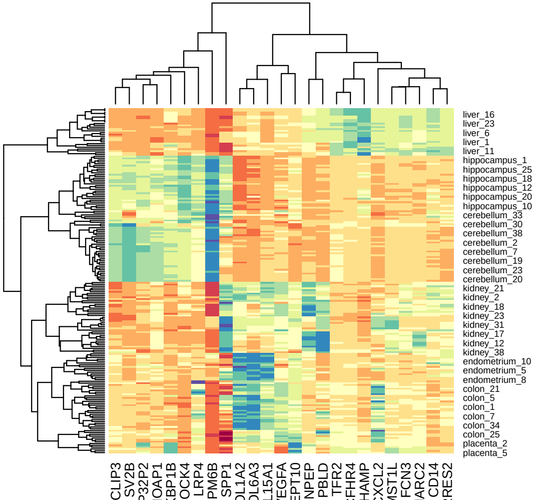
If the information about clusters in included in just a few features, including all the features can add enough noise that detecting clusters becomes challenging. One simple approach to try to remove features with no information is to only include those with high variance. In the movie example, a user with low variance in their ratings is not really informative: all the movies seem about the same to them.

Filtering features

For example, if we include only features (genes) with highest variance.

```r
library(matrixStats)
sds <- colSds(x, na.rm = TRUE)
o <- order(sds, decreasing = TRUE)[1:25]
heatmap(x[,o],
  col = RColorBrewer::brewer.pal(11, "Spectral"))
```

Filtering features



Session Info

```r
sessionInfo()
```

```
## R version 4.5.1 (2025-06-13)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sequoia 15.6.1
## 
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib 
## LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.12.1
## 
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
## 
## time zone: America/Denver
## tzcode source: internal
## 
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base     
## 
## other attached packages:
##  [1] matrixStats_1.5.0 dslabs_0.8.0      lubridate_1.9.4   forcats_1.0.0    
##  [5] stringr_1.5.1     dplyr_1.1.4       purrr_1.1.0       readr_2.1.5      
##  [9] tidyr_1.3.1       tibble_3.3.0      tidyverse_2.0.0   caret_7.0-1      
## [13] lattice_0.22-7    ggplot2_3.5.2    
## 
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.6         xfun_0.52            recipes_1.3.1
```