15 Unsupervised learning

In **supervised learning**, the data always have two parts, X and Y, and we focus on learning how to infer the relationship of X with Y. The outcome Y drives (or "supervises") the learning process. There are multiple inferential goals in surpervised learning. In machine learning, the inference has been mostly about *prediction*.

In **unsupervised learning**, there is no Y, or in other words, no distinction between X and Y. There are multiple goals in unsupervised learing tasks: (1) clustering, (2) density estimation, (3) dimensionality reduction, (4) encoding, (5) association rule learning, etc. Since there is no designated outcome to "supervise" the process, one could view all variables as X, or all variables as the outcome Y (with a constant X across all observations). Many methods for unsupervised learning rely on measures of dissimilarity or similarity, often in a pairwise fashion. Note that kernel SVMs rely on a pairwise similarity measure on X.

15.1 Clustering

Clustering is to divide data into non-overlapping subsets (i.e., a partition) of relatively homogeneous observations. Such a subset is called a **cluster**.

15.1.1 K-means clustering (ISLR 10.3.1)

For the K-means method, we need to have

(1) a measure of **dissimilarity**, $d(x_i, x_{i'})$, between any two points, x_i and $x_{i'}$;
(2) a definition of **centroid** for any set of data points;
(3) a number K, the target number of subsets.

Given a cluster, the within-cluster heterogeneity may be defined as

$$W(C) = \frac{1}{|C|} \sum_{i,i' \in C} d(x_i, x_{i'}), \tag{10.10}$$

where C is the set of indices for the cluster.

A popular choice for $d()$ is the **squared Euclidean distance** $d(x_i, x_{i'}) = \|x_i - x_{i'}\|^2 = \sum_j (x_{ij} - x_{i'j})^2$. This requires **all the features be standardized** unless you have a reason not to. The **centroid** of a cluster is often defined as the average $\frac{1}{|C|}\sum_{i\in C} x_i$. A centroid defined this way is the point x_0 that minimizes $\frac{1}{|C|}\sum_{i\in C}\|x_i - x_0\|^2$, its average squared Euclidean distance from all the observations in the cluster.

Goal: Given K, find a partition of data so that the total within-cluster heterogeneity defined in (10.10) is minimized. Given a partition of data, let $\{C_1, \ldots, C_K\}$ be the sets of indices for the partition. Then the goal is

$$\text{minimize}_{\{C_1,\ldots,C_K\}} \sum_{k=1}^{K} W(C_k). \tag{10.9}$$

***K*-means algorithm**: A generic version of the algorithm is below (shown in ISLR Figure 10.6):

1. Randomly assign the observations to K classes (every class needs to have at least one observation).
2. Iterate the following two steps:
   - Update centroids: Calculate the centroids for the K classes.
   - Update membership: Re-assign every observation to the class represented by the centroid "closest" to it (i.e., that centroid that has the smallest d from the observation).
3. Once converged (i.e., there is no change in membership or centroids), record the partition and its $\sum_k W(C_k)$.
4. Repeat 1–3 multiple times. Select the partition with the smallest $\sum_k W(C_k)$.

Notes:

- This is a greedy algorithm. Steps 1–3 do not guarantee to reach the overall mimimum. Step 4 increases the chance of reaching it.

- The algorithm can also be started with K centroids.
- Starting with K random centroids may lead to less than K clusters if one of the centroids is far away from all observations. Starting with a random partition with K non-empty subsets may also lead to less than K clusters. In `kmeans()`, K distinct observations are chosen as the starting centroids; this guarantees every cluster has at least one observation assigned to it.
- Adding features may bring more information (if they are informative features) or dilute information (if they are noise features). Adding noise features leads to an effect similar to the 'curse of dimensionality'.
- K is a hyperparameter. (I am not aware of any method to help select the K in K-means.)
- When $d()$ is the squared Euclidean distance, the iteration guarantees to reduce $\sum_k W(C_k)$ at every step. This is because for any cluster of m vectors $t_1, \ldots, t_m$, the within-cluster heterogeneity is

$$W = \frac{1}{m} \sum_{i,j} \|t_i - t_j\|^2 = 2 \sum_i \|t_i - \bar{t}\|^2,$$

where $\bar{t} = \frac{1}{n} \sum_i t_i$. After the m-th round of update, let $c_m : \{1, \ldots, n\} \to \{C_1, \ldots, C_K\}$ be the partitioning function and $\mu_{k,m}$ be the center of the k-th cluster. Then the total within-cluster heterogeneity for this partition is $T_m = 2 \sum_{i=1}^n \|x_i - \mu_{c_m(x_i),m}\|^2$. Changing memberships leads to $2 \sum_{i=1}^n \|x_i - \mu_{c_{m+1}(x_i),m}\|^2 < T_m$, and changing centroids leads to $T_{m+1} < 2 \sum_{i=1}^n \|x_i - \mu_{c_{m+1}(x_i),m}\|^2$. [Proof of the equation above for 1-dimension: On the one hand, $\sum_{i,j}(t_i - t_j)^2 = 2 \sum_{i<j}(t_i - t_j)^2 = 2[(m-1)\sum_i t_i^2 - 2\sum_{i<j} t_i t_j]$. On the other hand, $\sum_i (t_i - \bar{t})^2 = \frac{1}{m^2}[m(m-1)\sum_i t_i^2 - 2m \sum_{i<j} t_i t_j] = \frac{1}{m}[(m-1)\sum_i t_i^2 - 2\sum_{i<j} t_i t_j]$.]

The base R has a function `kmeans()` for performing the K-means clustering using squared Euclidean distance. Unfortunately it does not even have the option for scaling the features! So, we need to scale the features before calling `kmeans()`.

```r
library(ISLR)
Hitters1 = Hitters[,1:7]
names(Hitters1); dim(Hitters1)

aa = kmeans(Hitters1, 3, nstart=10)   ## K=3; run the algorithm 10 times
aa
str(aa)

all.equal(aa$tot.withinss, sum((Hitters1 - aa$centers[aa$cluster,])^2))  ## True
all.equal(aa$totss, sum((t(Hitters1) - apply(Hitters1,2,mean))^2))       ## True
```

Below we show why the algorithm should be run multiple times (specified with `nstart=`). We print out `tot.withinss` after every run of the algorithm. Note that every run has a different initialization. Partitions with different `tot.withinss` must be different partitions.

```r
for(ii in 1:20) {
  aa = kmeans(Hitters1, 3, nstart=1)
  print(aa$tot.withinss)
}
```

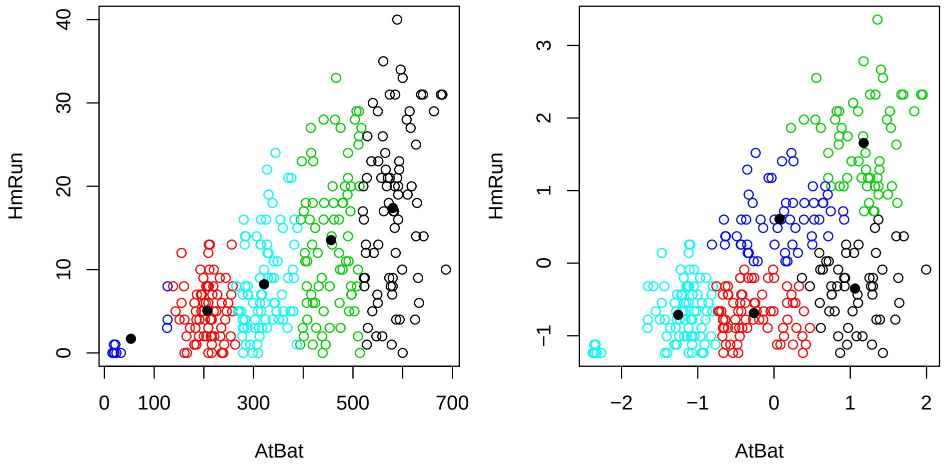
Now we standardize the features. The results are quite different between not scaling and scaling!

```r
Hitters2 = scale(Hitters1)
aa = kmeans(Hitters1, 3, nstart=10)
bb = kmeans(Hitters2, 3, nstart=10)
table(aa$cluster, bb$cluster)  ## quite different results!
```

We can use two features to demonstrate the difference between scaling and not scaling features. With no scaling, `AtBat` has a wider range (16–687) and thus a much higher impact than `HmRun` (range 0–40). With no scaling, we effectively assume every increment in `AtBat` is equivalent to every increment in `HmRun`.

```r
Hitters3 = Hitters1[,c(1,3)]
Hitters4 = scale(Hitters3)
aa = kmeans(Hitters3, 5, nstart=10)
bb = kmeans(Hitters4, 5, nstart=10)
```

```
table(aa$cluster, bb$cluster)

plot(Hitters3, col=aa$cluster); points(aa$centers, pch=19)
plot(Hitters4, col=bb$cluster); points(bb$centers, pch=19)
```



The **implementation** of the K-means algorithm in software often is not the algorithm above, but an optimized version that is faster and achieves exactly the same results as the one above. Optimizing the K-means algorithm is a classical topic in computer science.

A **clarification on various concepts**: It is helpful to distinguish these related concepts: (1) a goal, (2) a method, (3) an implementation algorithm of the method. For example,

(1) Our goal is to identify the partition that minimizes (10.9);
(2) The K-means algorithm is actually a method to achieve or approximate our goal;
(3) The steps as described in the method are an algorithm; this algorithm is often relatively easy to understand. The method in (2) may be achieved with different implemention algorithms. Some of them are more efficient (with respect to speed or memory usage) but may not be as easy to understand.

Sometimes a method is called an algorithm (as in here), which can cause confusions.

Similarly, for statistical models, it is helpful to distinguish these related concepts: (1) a model, (2) a model fitting criterion, (3) an algorithm to fit the model. For example,

(1) We wish to fit a linear model $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon$;
(2) We use the lasso criterion with $\lambda = 10$ (alternative criteria: least squares, maximum likelihood);
(3) We use gradient descent to fit the model (alternative algorithms: direct formula, Newton–Raphson).

The Keras syntax makes some of these distinctions, with its `layer` step mostly overlapping with (1) and (2), and its `compile` and `fit` steps overlapping with (3).

15.1.2 Hierarchical clustering (ISLR 10.3.2)

For the hierarchical clustering method, we need to have

(1) a measure of **dissimilarity**, $d(C_i, C_j)$, between any two non-overlapping *subsets*, C_i and C_j;
(2) either of the following:
   - a **threshold** for dissimilarity (a height in dendrogram) to determine if two subsets belong to the same cluster;
   - a number K, the target number of subsets.

Hierarchical clustering algorithm:

1. Start from n singletons (a singleton is a subset containing a single observation).
2. Repeat the following until a single set is reached:

- Among all the current subsets, merge the two subsets that have the smallest $d(C_i, C_j)$ (Because the smaller $d(C_i, C_j)$ the more similarity between C_i and C_j.)
- Record that $d(C_i, C_j)$.

3. Use a threshold value for $d(C_i, C_j)$ to cut the tree to obtain branches as clusters (as shown in ISLR Figure 10.9).

Notes:

- Hierarchical clustering is a *bottom-up* approach. In contrast, trees are built *top-down.*
- The results can be drawn as a **dendrogram**, with height $d(C_i, C_j)$ for the fuse point between C_i and C_j.
- The threshold value is a hyperparameter.
- ISLR Figures 10.10 and 10.11 illustrate this algorithm.

Definition of $d(C_i, C_j)$: The measure $d(C_i, C_j)$ may not be easily defined directly on all subsets. It is often easier to define a measure of dissimilarity between two points, $d(x_i, x_j)$. Then $d(C_i, C_j)$ can be defined through $d(x_i, x_j)$, in one of the following ways (called **linkage**):

- $d(C_1, C_2) = \max_{i \in C_1, j \in C_2} d(x_i, x_j)$ (**complete** linkage)
- $d(C_1, C_2) = \min_{i \in C_1, j \in C_2} d(x_i, x_j)$ (**single** linkage)
- $d(C_1, C_2) = \text{average}_{i \in C_1, j \in C_2} d(x_i, x_j)$ (**average** linkage)
- $d(C_1, C_2) = d(\text{centroid}_1, \text{centroid}_2)$ (**centroid** linkage)

Common choices for the definition of dissimilarity $d(x_i, x_j)$ are:

(1) Euclidean distance (often requiring *feature standardization* to make sense);
(2) One minus correlation coefficient (e.g., Pearson or Spearman correlation). The correlation is between two observations across all features, and so *feature standardization* is often necessary for the results to make sense.
(3) One minus Jaccard index, if the data are binary indicating presence or absence of features. The **Jaccard index** between two observations is (# features present in both) / (# features present in either). In R package `vegan`, `vegdist()` can calculate this and other dissimilarity indices.

Notes:

- Examples in ISLR Figure 10.12; also see below.
- Single linkage often has the signature pattern of adding one observation at a time.
- Different measures of dissimilarity can reflect quite differently on whether two observations are "similar" or not. ISLR Figure 10.13 shows an example for the definitions (1) and (2) above.

The base R has `hclust()` for hierarchical clustering, which by default uses complete linkage. It takes a `dist` object, which can be generated by calling `dist()` or `as.dist()`. We can use `cutree()` to cut a `hclust` tree into clusters. The function `dist()` by default computes the Euclidean distance between every pair of observations.

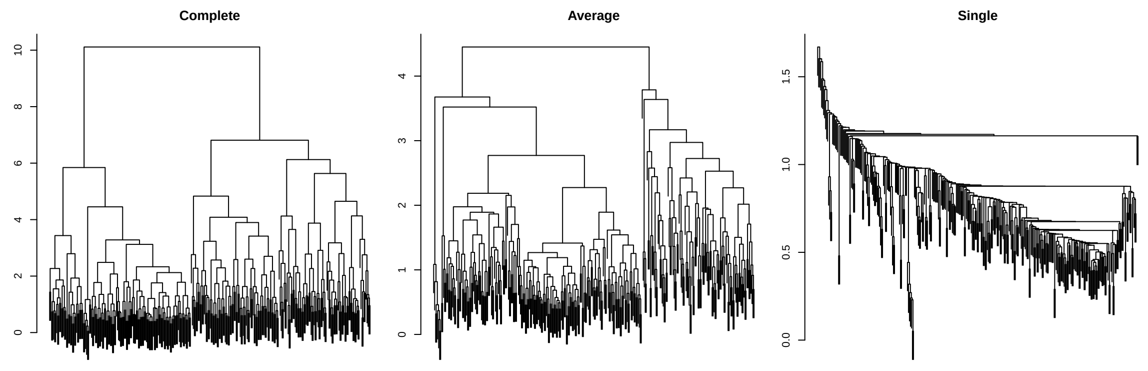
```r
library(ISLR)
Hitters1 = Hitters[,1:7]
Hitters2 = scale(Hitters1)  ## scaling the features
dist1 = dist(Hitters2)      ## pairwise distance
dim(Hitters2)               ## 322 x 7
str(dist1)                  ## 51681 = 322 * 321 / 2
sqrt(sum((Hitters2[1,]-Hitters2[2,])^2))  ## dist between 1st and 2nd observations
```

One can obtain a partition by specifying either the height in a dendrogram or the target number of subsets.

```r
cluster1 = cutree(hclust(dist1), h=5.7)   ## height 5.7
cluster2 = cutree(hclust(dist1), k=5)     ## K=5
table(cluster1); table(cluster2)
table(cluster1, cluster2)  ## same clustering
```

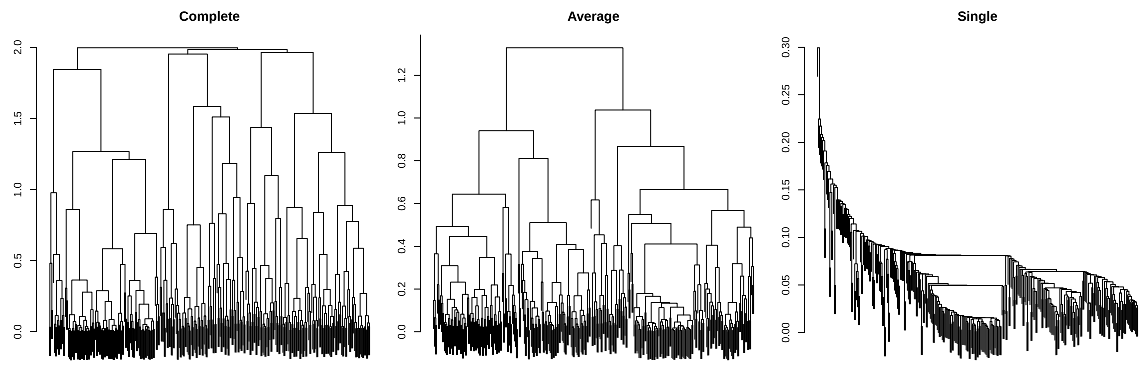
The dendrograms are:

```r
plot(hclust(dist1, method='complete'), labels=F, xlab='', main="Complete")
plot(hclust(dist1, method='average'), labels=F, xlab='', main="Average")
plot(hclust(dist1, method='single'), labels=F, xlab='', main="Single")
```



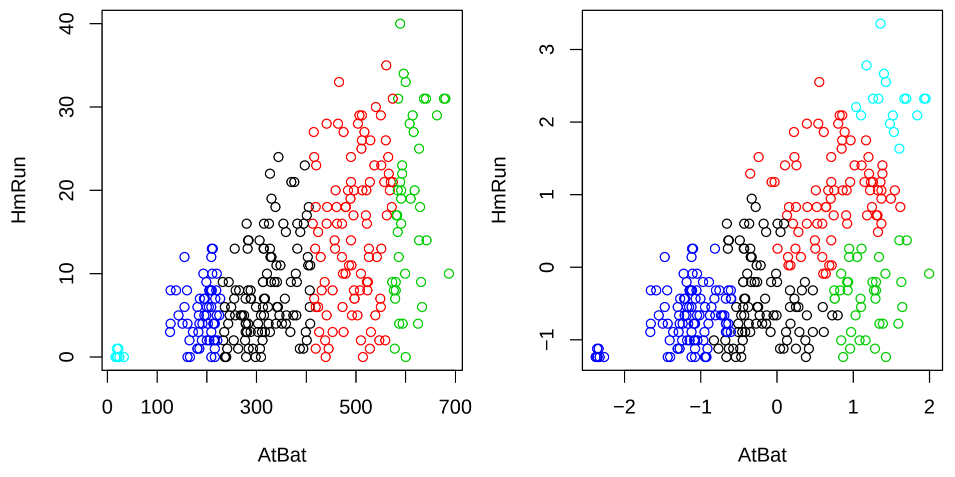
Now we define the dissimilarity as 1 - Pearson correlation, which has a range $[0, 2]$.

```
dist2 = as.dist(1 - cor(t(Hitters2)))

plot(hclust(dist2, method='complete'), labels=F, xlab='', main="Complete")
plot(hclust(dist2, method='average'), labels=F, xlab='', main="Average")
plot(hclust(dist2, method='single'), labels=F, xlab='', main="Single")
```



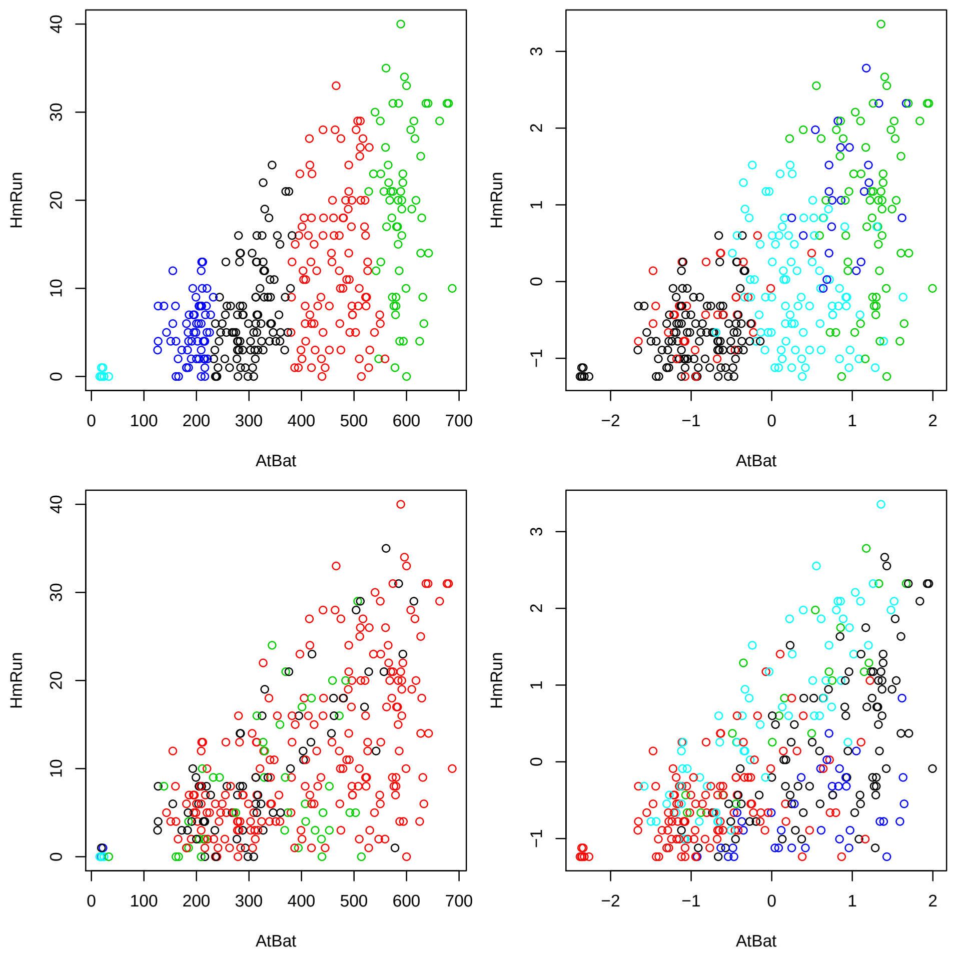
We now use two features to demonstrate the difference between scaling and not scaling features.

```
Hitters3 = Hitters1[,c(1,3)]
Hitters4 = scale(Hitters3)
aa = cutree(hclust(dist(Hitters3)), k=5)
bb = cutree(hclust(dist(Hitters4)), k=5)
table(aa, bb)
plot(Hitters3, col=aa); plot(Hitters4, col=bb)
```



Finally we use all 7 features and compare the results between Euclidean distance and 1-cor and between non-scaling and scaling. The results are quite different. We can plot the results with 2 features as the backdrop.

```
aa = cutree(hclust(dist(Hitters1)), 5)                ## no scaling, Euclidean
bb = cutree(hclust(dist(Hitters2)), 5)                ## scaled, Euclidean
cc = cutree(hclust(as.dist(1-cor(t(Hitters1)))), 5)   ## no scaling, 1-cor
dd = cutree(hclust(as.dist(1-cor(t(Hitters2)))), 5)   ## scaled, 1-cor
table(aa, cc); table(bb, dd)

plot(Hitters3, col=aa); plot(Hitters4, col=bb)
plot(Hitters3, col=cc); plot(Hitters4, col=dd)
```



15.1.3 Clustering using a mixture model

For the mixture model approach, we need to have

(1) a family of distributions (with parameters θ) for the clusters;
(2) a number K, the target of subsets.

In a **mixture model**, we assume the data are from a mixture of K distributions with unknown parameters θ_k and unknown mixture probabilities π_k:

$$h(y|\phi) = \sum_{k=1}^{K} \pi_k f(y|\theta_k) \quad \text{subject to} \quad \pi_k \geq 0, \quad \sum_{k=1}^{K} \pi_k = 1,$$

where $\phi = (\pi_1, \ldots, \pi_K, \theta_1, \ldots, \theta_K)$. Once we have estimated all the parameters, we can calculate the posterior probability for each observation to belong to a class k,

$$\hat{p}_k = P(k|y, \hat{\phi}) = \frac{\hat{\pi}_k f(y|\hat{\theta}_k)}{\sum_k \hat{\pi}_k f(y|\hat{\theta}_k)},$$

and assign the observation to the class with the largest posterior probability.

Notes:

- Because this is a likelihood-based method, AIC and BIC can be calculated to help select K.
- The R package `flexmix` has a function `stepFlexmix()` for this. See the next section for an example.
- This method is similar to the mixure model described for LDA/QDA. However, the latter is for supervised learning, with data that has known classes for the observations, and that often has known probabilities π_k.

This method is a special case of **latent class regression**, in which we assume that given x, the outcome y is from a finite mixture distribution with K components,

$$h(y|x, \phi) = \sum_{k=1}^{K} \pi_k f(y|x, \theta_k) \quad \text{subject to} \quad \pi_k \geq 0, \quad \sum_{k=1}^{K} \pi_k = 1.$$

When there are no x (or a constant x across all observations), **this regression problem becomes a clustering problem**! Again, we can calculate the posterior probability for each observation as

$$P(k|x, y, \hat{\phi}) = \frac{\hat{\pi}_k f(y|x, \hat{\theta}_k)}{\sum_k \hat{\pi}_k f(y|x, \hat{\theta}_k)},$$

and assign the observation to the class with the largest posterior probability.

For both methods, an **expectation-maximization** (EM) algorithm can be used to obtain the parameter estimates. EM algorithms are iterative, looping through an E-step and an M-step to update parameter estimates until convergence. Specifically, after initializing the parameters, we iterate between the following two steps:

- E-step: Given current $\hat{\phi}$, calculate $\hat{p}_{ik} = P(k|x_i, y_i, \hat{\phi})$. Then update $\hat{\pi}_k = \frac{1}{n}\sum_{i=1}^{n} \hat{p}_{ik}$ $(k = 1, \ldots, K)$.
- M-step: Maximize $l_k(\theta_k) = \sum_i \hat{p}_{ik} \log f(y_i|x_i, \theta_k)$ to obtain new $\hat{\theta}_k$ $(k = 1, \ldots, K)$.

[Math details: Consider the full data $\{(y_i, x_i, g_i)\}$, where g_i is the unobserved class for observation i. The log-likelihood for the full data is $l = \sum_i \log f(y_i|x_i, \theta_{g_i})$. The E-step is to calculate the expectation $l_E = E_{G|(Y,X)}(l)$ under the current estimate of the conditional distribution of $G|(Y, X)$. Then $l_E = \sum_i \sum_k \hat{p}_{ik} \log f(y_i|x_i, \theta_k) = \sum_k l_k(\theta_k)$. The M-step is to maximize $l_E = \sum_k l_k(\theta_k)$ with respect to all the parameters, which is equivalent to maximizing $l_k(\theta_k)$ for all k.]

15.1.4 Example: Clonal detection with cancer single-cell sequencing

Gawad et al. Dissecting the clonal origins of childhood acute lymphoblastic leukemia by single-cell genomics. PNAS. 2014 111:17947–17952. (paper; code)

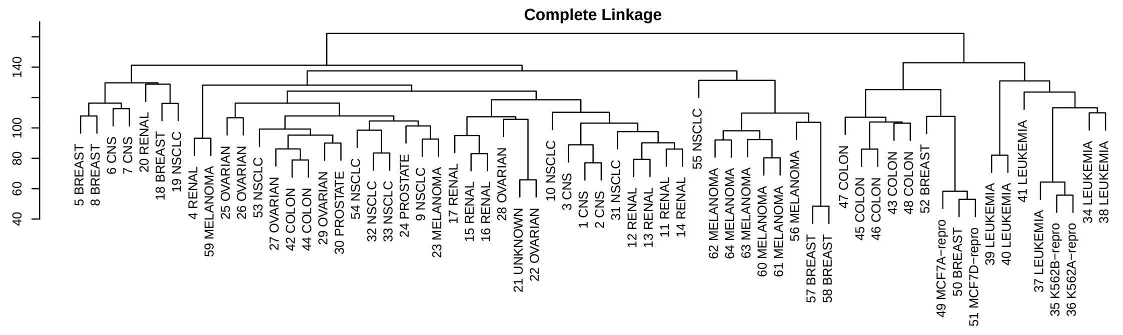
15.1.5 Example: NCI60 data

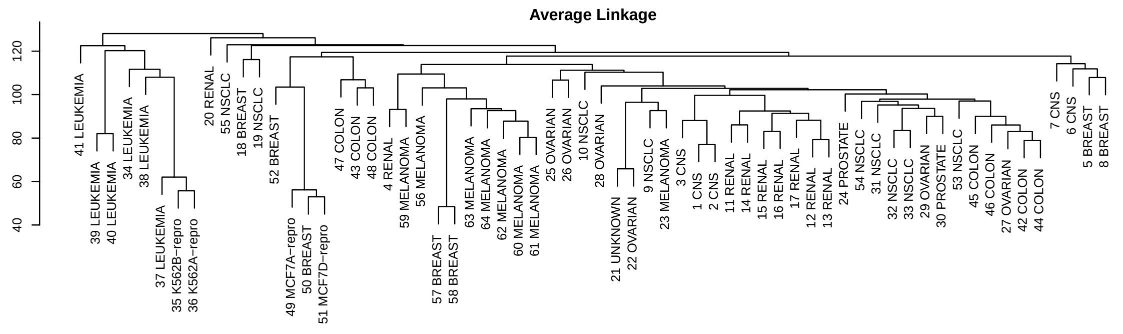
From Ross et al. Nature Genetics, 2000 24:227–234. See http://genome-www.stanford.edu/nci60/ for more details and some of the clustering plots.

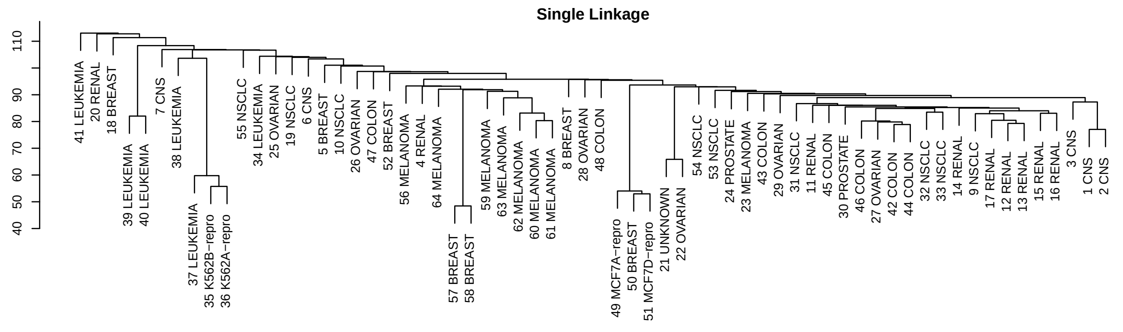
```r
library(ISLR)
nci.labs=NCI60$labs
nci.data=NCI60$data
nci.labs = paste(1:64, nci.labs)  ## add indices to labels for the plots
dim(nci.data)  ## 64 x 6830
hist(nci.data)

sd.data = scale(nci.data)  ## standardize the columns (genes)
data.dist = dist(sd.data)  ## default is Euclidean distance
```

We perform hierarchical clustering on the data.

```r
hc1 = hclust(data.dist)                    ## complete linkage by default
plot(hc1, labels=nci.labs, main="Complete Linkage", xlab="", ylab="", sub="")
#abline(h=hc1$height[1:3], col='grey')      ## add the first 3 heights

hc2 = hclust(data.dist, method="average")  ## average linkage
plot(hc2, labels=nci.labs, main="Average Linkage", xlab="", ylab="", sub="")

hc3 = hclust(data.dist, method="single")   ## single linkage
plot(hc3, labels=nci.labs, main="Single Linkage", xlab="", ylab="", sub="")
```







The information necessary for the plots are stored in the results.

```r
names(hc1)
hc1$order    ## the order of observations to be drawn
hc1$height
dim(hc1$merge); hc1$merge[1:10,]  ## singletons are negative; merged subsets are positive
```

15.1.6 Other clustering methods and R packages

There are several other clustering methods and several R packages for clustering. Here is an incomplete list.

- flexmix for admixture modeling.

- `NbClust`
- `GMD`
- Affinity propagation (AP) (Frey and Dueck, Science 2007, 315:972–976)
- X-means: An extension of K-means

15.2 Market basket analysis (ESL 14.2)

The goals of **market basket analysis** (MBA) are to mine:

- **Frequent item sets**: Find items that are frequently purchased together.
- **Association rules**: Find simple prediction models (called *association rules*) that have both good frequency (i.e., high support) and good prediction (i.e., high confidence and/or lift).

15.2.1 Data coding

Supermarket transaction data may contain billions of transactions (i.e., $n \sim 10^9$) and many features. There may be various levels of interest:

- Product categories (e.g., peanut butter), $p \sim 10^4$;
- Brands (e.g., Jif peanut butter), $p \sim 10^5$;
- UPCs (e.g., Jif Extra Crunchy 16 oz), $p \sim 10^6$.

The features can be coded in one of the following ways:

- Binary: $X_j = 1$ if item j is purchased in a transaction and $X_j = 0$ if not.
- Numerical: X_j is the number of item j purchased, or in number categories (e.g., “0”, “1”, “2-4”, “>4”).
  - This allows us to evaluate events like “2 or more Jif Extra Crunchy 16 oz are bought”.
  - *Conjunctive rules* may be used to simplify the problem: Only focus on events defined by $\bigcap_j (X_j \in s_j)$, where s_j is a subset of S_j, which is the set of all possible values for X_j. Under conjunctive rules, the scenario in the left panel of ESL Figure 14.1 will be ignored.
- One-hot/Dummy variable: For example, $Z_k = 1$ if “2-4” “Jif Extra Crunchy 16 oz” are purchased in a transaction; $Z_k = 0$ otherwise. (ESL Figure 14.1 middle and right panels)
  - This is equivalent to requiring that s_j either contains a single value or is the whole set S_j.

The one-hot coding is currently the most commonly used in market basket problems.

15.2.2 Frequent item sets

With one-hot coding, a transaction has an **item set**, which is a set defined as $K = \{Z_j : Z_j = 1\}$. For example, in the `Groceries` data, which has one-hot encoding for 169 product categories, an item set can be `{tropical fruit, yogurt, coffee}`. The **support** of an item set K is the probability that the items in K are bought together; it is denoted as $T(K) = \Pr(K)$.

If all items in A are also in B, A is a **subset** of B and B is a **superset** of A. Because they are item sets, A has a higher support than B. That is, $T(A) \geq T(B)$ when A is a subset of B. This fact is the basis of the Apriori algorithm. (Note that this is opposite from event sets in probability theory. Between two sets of events A and B, $\Pr(A) \leq \Pr(B)$ when A is a subset of B.)

An item set A is **frequent** at threshold t if $T(A) > t$. An item set is **maximally frequent** at threshold t if none of its immediate supersets is frequent at threshold t. An item set A is **closed** if none of its immediate supersets has the same support. A maximally frequent item set must be closed frequent, and a closed frequent item set must be frequent by definition.

15.2.3 Association rules

An **association rule** is just a rule $A \to B$, where the item sets A and B do not overlap. Here, A is the *antecedent* and B is the the *consequent*. Some example association rules are in ESL 14.2.3.

- The **support** of the rule is $T(A \to B) = T(A \text{ and } B) = \Pr(A \text{ and } B)$.
- The **confidence** of the rule is $C(A \to B) = T(A \text{ and } B)/T(A) = \Pr(B|A)$.
- The *expected confidence* is $T(B) = \Pr(B)$.
- The **lift** of the rule is $L(A \to B) = C(A \to B)/T(B) = \Pr(B|A)/\Pr(B) = \Pr(A \text{ and } B)/\Pr(A)\Pr(B)$.

Notes:

- confidence = support / $T(A)$
- lift = confidence / $T(B)$
- The lift of the rule $A \to B$ is the same as the lift of the rule $B \to A$.
- The lift is the ratio of confidence and expected confidence.
- The concept of lift is similar to the concept of "relative risk" in genetic epidemiology. For example, the sibling relative risk for type II diabetes is defined as the ratio of Pr(type II diabetes | a sibling has type II diabetes) and Pr(type II diabetes) in general population.
- The lift is the ratio of the joint probability of A and B in real data to that under independence. (A similar concept is the mutual information between two random variables.)

15.2.4 Association rule mining

Association rule mining: Identify all association rules that have "good" support, confidence and/or lift. Specifically, $T(A \to B) > t$, $C(A \to B) > c$ and/or $L(A \to B) > l$, where $t > 0$, $c > 0$ and $l \geq 1$.

Any association rule $A \to B$ with support $> t$ must have $T(K) > t$, where K is the union of the item sets A and B. Thus, to identify all association rules with support $T(A \to B) > t$, where $t > 0$, we first need to identify all candidate item sets K that have support $> t$. In other words, we need to first mine frequent item sets.

Let L_m be the set of all size-m item sets that have support $> t$. Then for any item set in L_m, its size-k subset $(k < m)$ must be a member of L_k.

Apriori algorithm (Agrawal and Srikant, 1994):

1. Search all singletons (item sets of size 1) to create L_1.
2. For $m = 2, 3, \ldots$, generate L_m:
   a. Join step: For any two sets $p, q \in L_{m-1}$ that share $m-2$ items, create their union c;
   b. Prune step: If c has a size-$(m-1)$ subset that is not in L_{m-1}, drop c;
   c. Check support: If c has support $> t$, keep it.
3. For every item set c with support $> t$, consider every possible split of c into two subsets, A and B, and if the confidence (and/or lift) of $A \to B$ is above a predetermined threshold, keep it.

Here the first two steps are to mine frequent item sets, and the third step is to mine association rules.

Alternative algorithms:

- The **Eclat algorithm** (Zaki, 2000) uses the transaction ID (`tid`) representation of data. A tid-list for an item set is the list of IDs whose transactions support the item set.
- The **FP-growth** algorithm (Han, Pei, Yin, 2000) uses the FP-tree representation of data.

15.2.5 An example

R has package `arules` for mining association rules. The package `arulesViz` provide visualization tools. The `arules` package comes with a small dataset `Groceries`, in which the features are 169 product categories in one-hot encoding. The `Groceries` dataset has class `transactions`. It has three "slots": `data`, `itemInfo`, and `itemsetInfo`.

```
library(arules)
?Groceries
data(Groceries)                    ## load the dataset
Groceries                          ## print very little information
str(Groceries)                     ## @data is a sparse matrix; @itemInfo is a data frame
```

The slot `itemInfo` is a data frame that contains information on item categories in three levels. The first element of `itemInfo` must have name `label` and it contains the item labels.

```
itemLabels(Groceries)                ## 169 product categories
itemInfo(Groceries)                  ## same as Groceries@itemInfo , a data frame
dim(Groceries@itemInfo)              ## 169 x 3
head(Groceries@itemInfo)
unique(Groceries@itemInfo$labels)   ## 169 unique values in labels
unique(Groceries@itemInfo$level2)   ## 55 categories in level 2
unique(Groceries@itemInfo$level1)   ## 10 categories in level 1
```

The slot `data` is a sparse matrix and it contains 9,835 transactions. To view the individual records, use `inspect()`, `size()`, `LIST()`, on an individual transaction or a set of transactions.

```
summary(Groceries)
dim(Groceries)                       ## 9835 x 169; dim(Groceries@data) gives 169 x 9835
length(Groceries)                    ## 9835

inspect(Groceries[2])
inspect(head(Groceries, 3))
size(head(Groceries, 3))
LIST(head(Groceries, 3))

table(size(Groceries))               ## distribution of transaction size
mean(size(Groceries))                ## average size 4.409 (2.6% of 169 categories)
```

Frequencies at which the categories were bought. Barplot of the most frequent categories.

```
itemFrequency(Groceries)                                        ## relative freq (probability)
sort(itemFrequency(Groceries, type="absolute"), decreasing=T)  ## frequency in decreasing order

library(magrittr)                                               ## alternative ways
Groceries %>% itemFrequency(type="absolute") %>% sort(decreasing=T)
Groceries %>% LIST %>% unlist %>% table %>% sort(decreasing=T)

itemFrequencyPlot(Groceries, main="Relative Frequency")
itemFrequencyPlot(Groceries, topN=10, type="absolute", main="Top 10 Item Frequency")
itemFrequencyPlot(Groceries, topN=10, main="Top 10 Item Relative Frequency")
```

Mining frequent item sets: Use either `eclat()` or `apriori()`. Eclat and Apriori are different algorithms. To turn off the printout of these functions, specify `control=list(verbose=F)`. The threshold values for item set selection are specified in `parameter=list()`. See `help("APparameter")` for all the parameters one can specify.

```
freqitems = eclat(Groceries, parameter=list(supp=0.05, maxlen=10))
#freqitems = eclat(Groceries, parameter=list(supp=0.05, maxlen=10, minlen=2))
freqitems                 ## print very little information
str(freqitems)            ## @items has the selected item sets; @quality has the metrics

summary(freqitems)
freqitems %>% head %>% inspect            ## the first several item sets
freqitems %>% sort %>% head %>% inspect   ## item sets with largest support
```

The same results can be obtained by using `apriori()` with option `target="frequent itemsets"`. The selected item sets may have a different order than that from `eclat()`.

```
freqitems2 = apriori(Groceries, parameter=list(supp=0.05, maxlen=10, target="frequent itemsets"))
aa = inspect(sort(freqitems))
bb = inspect(sort(freqitems2))
all.equal(aa, bb)               ## TRUE, indicating the results are same after sorting
```

For "maximally frequent itemsets", only 27 item sets are selected, compared to 31 "frequent itemsets". The 4 missing itemsets are `{whole milk}`, `{other vegetables}`, `{rolls/buns}`, and `{yogurt}`. They are in the 3 order-2 "frequent itemsets" and thus are not maximally frequent.

```
freqitems3 = apriori(Groceries, parameter=list(supp=0.05, maxlen=10,
                                               target="maximally frequent itemsets"))
freqitems3    ## 27 itemsets compared to 31 above
freqitems2 %>% sort %>% head %>% inspect
freqitems3 %>% sort %>% head %>% inspect
```

To plot the support vs. order for a set of item sets.

```
library(arulesViz)
plot(freqitems)
```

Mining association rules: Use `apriori()` or `ruleInduction()`. It is nice to see all parameter values are spelled out in the output, no matter whether they are specified by the user or the default values.

```
rules1 = apriori(Groceries, parameter=list(supp=0.01, conf=0.5, maxlen=10))
inspect(rules1)   ## lhs, rhs, support, confidence, lift, count
plot(rules1)      ## support vs. confidence, colored according to lift

rules1 %>% sort(by="support") %>% head %>% inspect
rules1 %>% sort(by="confidence") %>% head %>% inspect
rules1 %>% sort(by="lift") %>% head %>% inspect
```

We can request to output only the rules that lead to buying `whole milk` or rules that start from buying 'whole milk'. Notice the use of `minlen=2` to avoid rules such as $\emptyset \to A$.

```
rules2 = apriori(Groceries, parameter=list(supp=0.002, conf=0.8),
                 appearance=list(rhs="whole milk"))
rules3 = apriori(Groceries, parameter=list(supp=0.03, conf=0.1, minlen=2),
                 appearance=list(lhs="whole milk"))

inspect(rules2)
inspect(rules3)
plot(rules2, jitter=1); plot(rules3)
```

Note that since confidence = support / $T(A)$, the plot of confidence against support is a straight line when all the rules have the same antecedent A, as in `rules3` above.

15.2.6 Other datasets

Market basket analysis can be used to analyze non-transactions data. For example, the `arules` package has the `Adult` dataset, which is the UCI `Adult` data in the "transactions" format. For one-hot encoding, numerical variables were converted to categories; e.g., `age` has been converted to 4 categories: Young, Middle-aged, Senior, Old. There are 48,842 records in 115 one-hot variables.

The original data are in `AdultUCI`. The variables `fnlwgt` and `education-num` in `AdultUCI` are not in `Adult`.

```
library(arules)
data("Adult"); data("AdultUCI")
str(Adult)                        ## in "transactions" format
str(AdultUCI)                     ## original data as a data frame
LIST(Adult[1])                    ## first person (transactions format)
AdultUCI[1,]                      ## first person (original data)
table(Adult@itemInfo$variables)   ## number of categories in each of the 13 variables
```

Because this data was converted from a regular dataset, most observations have the same number of "items". Only those with missing data will have fewer "items". This is different from real transactions data.

```
table(size(Adult))              ## those with size<13 have missing data
which(size(Adult) == 9)         ## take a look at an observation with only 9 one-hot variables
LIST(Adult[34722])
AdultUCI[34722,]                ## 4 variables have missing data
```

Other datasets in the `arules` package.

```
data(package="arules")
```

Here is another demonstration (code) of MBA using the Online Retail Data Set.

15.3 Density estimation and generative models in high dimensions

In machine learning, models for learning the distribution behind data so that we can generate new instances that appear legitimate are called **generative models**. (In contrast, models for prediction are called **discriminative models**.) Generative models may have an explicit model for the distribution (i.e., density estimation), or may just be an algorithm for generating new instances. The methods fall in two broad types:

- Specify a model (i.e., a family of distributions indexed by parameters), and estimate the parameters by maximizing the likelihood L, or equivalently, minimizing the cost $C = -\log L$. This is the typical approach in classical statistics. For high-dimensional data, a popular method is the restricted Boltzmann machines (RBMs).
- Transform the problem to a supervised learning problem and apply a wide variety of supervised learning tools to attack the problem. We describe two approaches below.

15.3.1 Learning against a reference distribution (ESL 14.2.4)

Goal: Suppose we have a dataset with N observations from an unknown distribution G. Let $g(x)$ be the density function of distribution G. Our goal is to estimate $g(x)$.

Rationale: Let $g_0(x)$ be the density function of a known distribution G_0 over the same sample space. For example, G_0 could be a uniform distribution over the sample space. Consider a mixture distribution M with density function $(g(x) + g_0(x))/2$. If we know $\mu(x) = \frac{g(x)}{g(x)+g_0(x)}$, then we can derive $g(x) = g_0(x)\frac{\mu(x)}{1-\mu(x)}$. Note that $\mu(x)$ is the conditional probability that an observation from distribution M belongs to distribution G.

Approach: To learn $\mu(x)$, we can simulate a large number, N_0, of observations from $g_0(x)$, and then assign weight $N_0/(N + N_0)$ to each of the N observations in the real data, and weight $N/(N + N_0)$ to each of the N_0 observations in the simulated data. The pooled dataset has equal total weight from the two distributions. We assign $Y = 1$ to those in the real data and $Y = 0$ to those in the simulated data. Now we can apply machine learning tools to estimate $\mu(x) = \Pr(Y = 1|x)$. ESL Figure 14.3 shows an example of using a semiparametric logistic regression model to learn the density of a dataset.

Choice of the reference: In this approach, if $\mu(x)$ cannot be accurately estimated, nor is $g(x)$. This can happen when $g_0(x)$ is very different from $g(x)$. Just setting G_0 to be a uniform distribution may not be an effective choice. Ideally one should choose the distribution G_0 to be close to G. But this may not be achievable in practice.

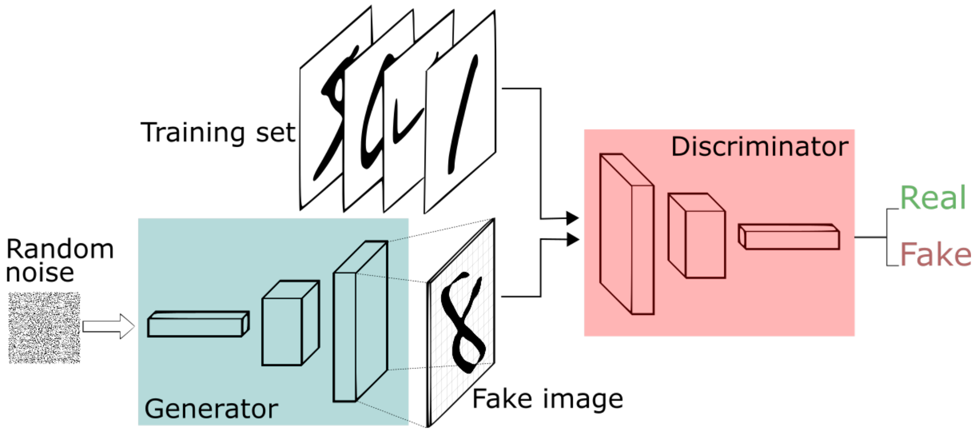
15.3.2 Generative adversarial networks

Generative adversarial networks (GANs) (Goodfellow et al., 2014) are designed for high-dimensional data. Here, an unsupervised learning problem is transformed to a sequence of supervised learning problems. Instead of specifying a reference distribution as in the approach above, we develop a reference by learning it!

The method has two components (diagram below from https://deeplearning4j.org/generative-adversarial-network):

- The **generator** is a model $G_\theta(z)$ (with parameters θ) for generating fake data $x_0 = G(z)$, where z is from a known distribution (e.g., a uniform distribution; that is, “random noise” in the diagram).

- The **discriminator** is a model $D_\delta(x)$ (with parameters δ) learned from the data $\{(x_1,1),\dots,(x_N,1)\}$ and $\{(x_{01},0),\dots,(x_{0N},0)\}$. Here, $D_\delta(x)$ is the probability that x is real data. The function $D_\delta(x)$ can serve as a score for every x, and this score will be used to drive the learning of G_θ.



The method works in the following way:

1. Initialize the generator G_θ with $\theta = \theta_0$.
2. Iterate between the following two steps (both are supervised learning tasks):
   - Given the current function G_θ, generate fake data $\{x_{01},\dots,x_{0N}\}$. Train the discriminator $D_\delta(x)$ on $\{(x_1,1),\dots,(x_N,1)\}$ and $\{(x_{01},0),\dots,(x_{0N},0)\}$.
   - Given the current function D_δ, train the generator to maximize $D_\delta(G_\theta(z))$ over θ, or equivalently, to minimize the cost $\log(1 - D_\delta(G_\theta(z)))$.

Goodfellow et al., (2014) nicely describes the idea behind GANs: "In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles."

Goodfellow et al. (2014) also describes a stochastic version of the algorithm to be used for large datasets.

An interesting application is the generation of fake human faces (video) by a Karras et al., 2018, a team from NVIDIA who trained a GAN progressively using 30,000 pictures of celebrities.

Since the publication of the original GAN paper, many variants of GANs have been developed. Here is an example of applying a variant of GAN, Auxiliary Classifier GAN (ACGAN), to the MNIST dataset using the R `keras` package.

15.4 Multi-dimensional scaling (ESL 14.8)

MDS and SOM are dimension reduction techniques, mostly for the purpose of visualization. Unlike PCA, the target dimension needs to be specified in MDS and SOM.

In multi-dimensional scaling (MDS), we seek to identify a low-dimensional representation (called a **configuration**), $\{z_1,\dots,z_n\}$ in R^k, where k is often 2 or 3, of the original data such that their pairwise dissimilarity (or similarity) matrix $M' = \{d'_{ij}\}$ is as close as possible to that for the original data, $M = \{d_{ij}\}$. For similarity, we use the notation $M' = \{s'_{ij}\}$ and $M = \{s_{ij}\}$.

Areas of applications of MDS:

- *Proximity data* (e.g., pairwise similarity or dissimilarity, confusion matrix);
- Visualization of data (as an alternative to principal components) ($k = 2,3$);
- Visualization of networks ($k = 2,3$);
- In chemistry, identification of the spatial structure of a molecule with a known sequence ($k = 3$).

For MDS, we need to define:

1. a measure of pairwise dissimilarity d_{ij} (or similarity s_{ij}) between observations in the original data;
2. a measure of pairwise dissimilarity d'_{ij} (or similarity s'_{ij}) in R^k;
3. a **stress function**, which is a measure of closeness between matrices M and M'.

The goal of MDS is to identify a k-dimensional representation that minimizes the stress. Gradient descent is often used for this purpose.

15.4.1 Classical metric scaling

The classical metric scaling uses similarity. The measure of similarity in R^k is defined as centered inner product $s'_{ij} = \langle z_i - \bar{z}, z_j - \bar{z}\rangle$. The stress between $M' = \{s'_{ij}\}$ and $M = \{s_{ij}\}$ is $S_M = \sum_{i \neq j}(s_{ij} - s'_{ij})^2$.

Notes:

- If the similarity measure for the original data is also centered inner product $s_{ij} = \langle x_i - \bar{x}, x_j - \bar{x}\rangle$, the classical scaling is effectively the same as principal components analysis.
- Suppose the data is already centered. Then $s((9,0),(1,1)) = s((3,0),(3,3))$. These two pairs of vectors have the same "similarity" as measured by inner product, but their distances are quite different.
- This method is also called the *Principal Coordinate Analysis* in ecology.

The base R has a function `cmdscale()` for the classifical MDS. We use the NCI60 dataset as an example. We use two versions of the data: all data, and those with known cancer types in their labels.

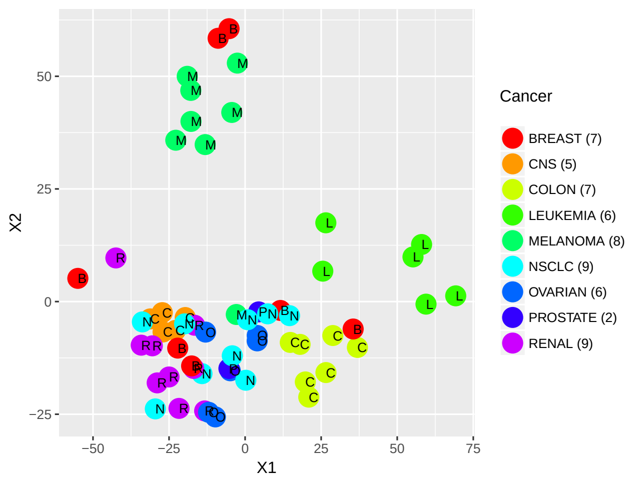
```r
library(ISLR)
nci.labs = NCI60$labs
nci.data = NCI60$data
dim(nci.data)    ## 64 x 6830
table(nci.labs)

ncitypes = c("BREAST","CNS","COLON","LEUKEMIA","MELANOMA","NSCLC","OVARIAN","PROSTATE","RENAL")
nci.data2 = NCI60$data[NCI60$labs %in% ncitypes, ]
nci.labs2 = NCI60$labs[NCI60$labs %in% ncitypes]
dim(nci.data2)   ## 59 x 6830

aa = table(nci.labs2)
cancerlabels = paste(names(aa), ' (', aa, ')', sep='')   ## labels for the plots
```

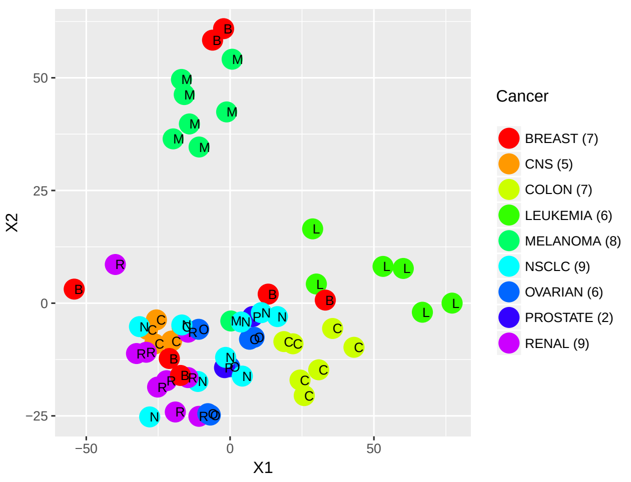
The function `cmdscale()` takes a "dist" object or a symmetric dissimilarity matrix. (Internally, a dissimilarity matrix is first converted to an inner product matrix.) We first analyze the whole dataset.

```r
data.dist1 = dist(scale(nci.data))
cmds2a = cmdscale(data.dist1, k=2, add=T, list.=T)  ## 2-dim, returning a list because list.=T
str(cmds2a)

## we only draw those with known cancer types
aa2a = subset(data.frame(cmds2a$points), nci.labs %in% ncitypes)
library(ggplot2)
ggplot(aa2a, aes(X1, X2)) +
  geom_point(aes(colour=nci.labs2), size=6) +
  labs(color="Cancer\n") +
  scale_color_manual(labels=cancerlabels, values=rainbow(10)[1:9]) +
  geom_text(aes(label=substr(nci.labs2, 1, 1)), hjust=0, size=3)
```

Now we repeat the analysis using the 59 samples with known cancer types.

```r
data.dist2 = dist(scale(nci.data2))
cmds2b = cmdscale(data.dist2, k=2, add=T, list.=T)
aa2b = data.frame(cmds2b$points)
## for the plot, replace aa2a by aa2b in ggplot() above.
```





Note that the sample labels were not used in the MDS analyses. They are shown in the plots to help visualize how well the MDS results represent the original data.

The 3D classical MDS results seem to show a slightly better separation for some cancer types.

```
cmds3b = cmdscale(data.dist2, k=3, add=T, list.=T)
x = cmds3b$points[,1]
y = cmds3b$points[,2]
z = cmds3b$points[,3]

library(rgl)
plot3d(x, y, z, size=5, col=as.numeric(factor(nci.labs2)))
text3d(x, y, z, nci.labs2, cex=.7)
```

15.4.2 Alternative metric scaling methods

Kruskal–Shepard metric scaling uses dissimilarity and stress $\sqrt{\sum_{i\neq j}(d_{ij}-d'_{ij})^2}$.

- It is also called *least squares scaling*.
- Square root does not matter because $\sqrt{\sum_{i\neq j}(d_{ij}-d'_{ij})^2}$ is minimized whenever $\sum_{i\neq j}(d_{ij}-d'_{ij})^2$ is minimized.

Sammon mapping uses dissimilarity and stress $\sum_{i<j}\frac{(d_{ij}-d'_{ij})^2}{d_{ij}}$.

- For a pair of observations with a small d_{ij}, the stress $\frac{(d_{ij}-d'_{ij})^2}{d_{ij}}$ is relatively larger than that in the least squares scaling. As a result, d'_{ij} is pushed to be close to d_{ij} even when d_{ij} is small.

Thus, because of how the stress functions are defined,

- In the classical and least squares scaling, the results are driven mostly by large values of s_{ij} or d_{ij}.
- In Sammon mapping, the results are driven by both large and small values of d_{ij}.

15.4.3 Non-metric scaling

Shepard–Kruskal non-metric MDS: The stress is

$$\sqrt{\frac{\sum_{i<j}[\theta(d_{ij})-d'_{ij}]^2}{\sum_{i<j}d'^2_{ij}}}.$$

Here, we do not focus directly on the original dissimilarity values d_{ij}. Instead, we allow *any* monotonic function of them, $\theta(d_{ij})$. Because an unknown function $\theta(d_{ij})$ is involved, the stress needs to be "normalized".

Sammon non-metric MDS: The stress is $\sum_{i<j} \frac{(d_{ij}-d'_{ij})^2}{d_{ij}} / \sum_{i<j} d_{ij}$.

The R `MASS` library has functions `isoMDS()`, `Shepard()`, and `sammon()` for Kruskal–Shepard and Sammon non-metric MDS. For the NCI60 dataset, `isoMDS()` gives very similar results as `cmdscale()`.

```
library(MASS)
mds2 = isoMDS(data.dist1, k=2)

mds2a = subset(data.frame(mds2$points), nci.labs %in% ncitypes)
ggplot(mds2a, aes(X1, X2)) +
  geom_point(aes(colour=nci.labs2), size=6) +
  labs(color="Cancer\n") +
  scale_color_manual(labels=cancerlabels, values=rainbow(10)[1:9]) +
  geom_text(aes(label=substr(nci.labs2, 1, 1)), hjust=0, size=3)

## Now the 3D MDS
mds3 = isoMDS(data.dist1, k=3)
library(rgl)
x = mds3$points[,1]
y = mds3$points[,2]
z = mds3$points[,3]
plot3d(x, y, z, size=5, col=as.numeric(factor(nci.labs2)))
text3d(x, y, z, nci.labs, cex=.7)
```

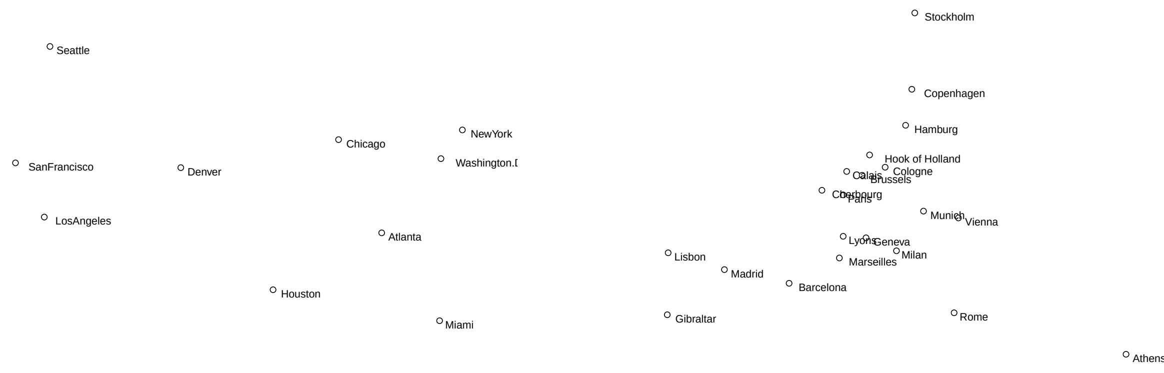
15.4.4 Example: Pairwise distances between cities

R has datasets `UScitiesD` and `eurodist`, which are "dist" objects containing pairwise distances between some cities in the US and Europe. According to their help page, "`eurodist` gives the road distances (in km) between 21 cities in Europe," and "`UScitiesD` gives 'straight line' distances between 10 cities in the US." This example shows how much 2D relative positions can be recovered by the classical MDS.

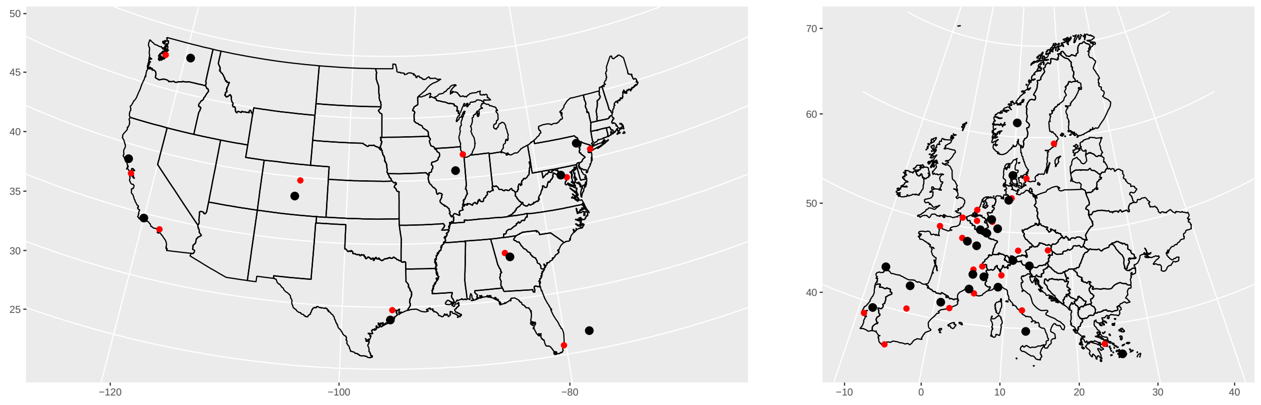
Because flipping and rotation do not change centered inner products nor pairwise distances, the results may be flipped or rotated or both. We first define a 2D rotation transformation given an angle in degrees counterclockwise.

```
rotate = function(angle) {
  aa = angle/180*pi
  matrix(c(cos(aa), -sin(aa), sin(aa), cos(aa)), 2)
}
```

```
USdist.cmds = cmdscale(UScitiesD)
USdist.cmds = -USdist.cmds                    ## flip both coordinates
USdist.cmds = USdist.cmds %*% rotate(-10)     ## rotate 10 degrees clockwise
plot(USdist.cmds, xlab="", ylab="", asp=1, axes=F, xlim=c(-1425, 1350))
text(USdist.cmds, rownames(USdist.cmds), adj=c(-.2, 1), cex=.8)

eurodist.cmds = cmdscale(eurodist)
eurodist.cmds[,2] = -eurodist.cmds[,2]        ## flip the latitude coordinate
eurodist.cmds = eurodist.cmds %*% rotate(10)  ## rotate 10 degrees counterclockwise
plot(eurodist.cmds, xlab="", ylab="", asp=1, axes=F, xlim=c(-1900, 2800))
text(eurodist.cmds, rownames(eurodist.cmds), adj=c(-.2, 1), cex=.8)
```



Now we put the MDS results on the real maps, with the correct city locations in red dots and the rotated MDS coordinates in black dots. The results for European cities are not as good probably because the distances are road distances. The distances for US cities are straight line distances. Note that the "spherical" straight line distance between two cities is often shorter than their distance on a 2D map. This effect is stronger when the two cities are closer to the poles.

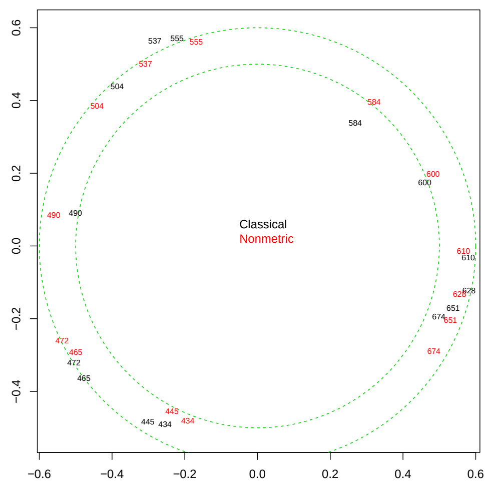


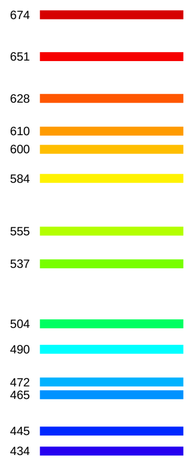
15.4.5 Example: Ekman data

The Ekman data contains average perceived similarity between 14 colors. It is available in the R `smacof` package. The help page of `ekman` has more details. We apply MDS methods to obtain a 2-dimensional representation of the 14 colors. The wavelengths of the 14 colors cover most of the spectrum of a full rainbow. The surprisingly good circular shape of the MDS results indicates that the methods yield good representations.

```r
library(smacof)
library(MASS)
ekman
ekmandis = sim2diss(ekman, method=1)  ## convert to dissimilarity = 1 - similarity

ekman.cmds2 = cmdscale(ekmandis, k=2, add=T, list.=T)
ekman.nmds2 = isoMDS(ekmandis, k=2)

## Draw the MDS results
plot(ekman.cmds2$points, type="n", asp=1, xlab="", ylab="")
text(ekman.cmds2$points, labels=rownames(ekman.cmds2$points), cex=.7)
text(ekman.nmds2$points, labels=rownames(ekman.nmds2$points), cex=.7, col=2)
legend("center", text.col=1:2, legend=c('Classical', 'Nonmetric'), bty='n')

## Draw circles for reference
theta = seq(0, 2*pi, length=180)
for(i in c(0.5, 0.6)) lines(x=i*cos(theta), y=i*sin(theta), lty=2, col=3)
```

```
## Draw the 14 colors
source("wavelength_to_rgb.r") ## From https://gist.github.com/friendly/67a7df339aa999e2bcfcfec88311abfc
aa = as.numeric(rownames(ekman.cmds2$points))
wavecol = NULL
for(i in 1:length(aa)) wavecol[i] = wavelength_to_rgb(aa[i])

plot(rep(0, length(aa)), aa, axes=F, xlab='', ylab='', type='n', xlim=c(-.2, 1))
segments(rep(0, length(aa)), aa, rep(1, length(aa)), aa, col=wavecol, lwd=12, lend=1)
text(rep(0, length(aa)), aa, adj=1.5, labels=aa)
```





15.4.6 Extensions of MDS (ESL 14.9)

ESL Figure 14.44 shows a scenario where the traditional MDS may fail. There are a few extensions of MDS.

Local MDS (Chen and Buja, 2008): Its stress function is (14.106). The focus is on small d_{ij}'s. An example is shown in ESL Figure 14.44.

Local linear embedding (LLE) (Roweis and Saul, 2000): Every observation has a "local structure" with its neighboring observations. Find a low-dimensional representation that keeps this local structure. ESL Figure 14.45 shows an application of LLE to 1,965 pictures of faces in a resolution of 20×28.

Isometric feature mapping (ISOMAP) (Tenenbaum et al., 2000)

15.4.7 Notes on MDS

In addition to the MDS functions above,

- The R package `smacof` provides a suite of functions for MDS. It also has some classical proximity datasets: `ekman`, `morse`, `winedat`, etc.
- The R package `vegan` has functions `monoMDS()` and `wcmdscale()`.
- GGobi has a plugin, GGvis, for MDS. But its R version `rggobi` seems not to have the MDS features. The R package `ggvis` is for interactive graphics and is not relevant to `rggobi`. [In the names "GGobi" and "GGvis" the first "G" refers to GTK. Earlier versions were called XGobi and XGvis, with "X" referring to the X Window system. The "gg" in `ggplot2` and many other related names refers to Wilkinson's Grammar of Graphics.]

15.5 Self-organizing maps (ESL 14.4)

Self-organizing maps (SOMs) are another method for dimension reduction and visualization. The online version (i.e., process data one at a time) of the SOM algorithm is:

- Start with a grid of **prototypes** in R^p, where p is the number of features. Let m_j be the initial position of prototype l_j.
- Let data gradually pull the prototypes towards them. Process data one by one (online algorithm).
  - For observation x_i, find the prototype m_j that is closest to it, move m_j and its "neighbors" m_k towards x_i. A simple version is in (14.46): $m_k \leftarrow m_k + a(x_i - m_k)$, where a is the *learning rate*. (14.47) is a slightly more sophisticated version.

Notes:

- The initial positions of the prototypes can be on the hyperplane of the first 2 principal components. The number of prototypes is often high. A 2D grid can be rectangular or hexagonal (triangular).
- Neighboring prototypes are defined according to their positions on the original grid, not on m_j's.
- As we move along, the learning rate and the neighbor "radius" may be gradually reduced (similar to simulated annealing).
- Using the Euclidean distance requires feature standardization to make sense.
- The result can be influenced by the order of observations. Online algorithms are also sensitive to outliers.
- Presenting the final result in the original grid can be misleading because the final locations of the prototypes can be highly disfigured, as shown in ESL Figure 14.17.
- If the neighbor of a prototype is always itself, this algorithm becomes an online version of K-means clustering.

Examples:

- ESL Figures 14.15–14.17. Note that the size of the grid is $25 \gg 3$, where 3 is the true number of groups.
- ESL Figure 14.19. The data are 12,088 articles in a newsgroup. The data are first processed into a *term-document matrix*, which is often very sparse.

The **batch version of SOM** is similar to K-means clustering. Iterate the following:

1. map all observations to their closest prototypes;
2. update prototype m_k by a weighted average of x_i that are mapped to the neighborhood prototypes of m_k.

If the neighbor of a prototype is always itself, then this batch version becomes the K-means clustering algorithm. In general, SOM can be viewed as a constrained version of K-means clustering.

R packages for SOM:

- `kohonen`
  - https://www.r-bloggers.com/self-organising-maps-for-customer-segmentation-using-r/
  - SOM applied to NBA data: https://clarkdatalabs.github.io/soms/SOM_NBA
- `som` (latest version in 2016)

15.6 R packages for visualization

For static graphics:

- `ggplot2`. The "gg" stands for "The Grammar of Graphics", the title of a book by Leland Wilkenson.

- `ggmap` is for drawing maps. Some demonstrations are here and here.

For interactive apps and graphics displayed in web browers:

- `shiny` is for building interactive web **apps**. It can work together with CSS themes, htmlwidgets, and JavaScript actions. Its gallery.

- Here is a list of **htmlwidgets**. One half of them are on CRAN. According to Joe Cheng: "Anything that's rendered into a rectangle of HTML/CSS/JavaScript, and needs to work 'everywhere' (console, Shiny, R Markdown, RStudio viewer, saved as .html file) should be an htmlwidget. `ggvis` isn't, only because it predates htmlwidgets." For example, `ggiraph` is an htmlwidget for animating ggplot2 graphics. It is maintained here.

- `ggvis` is an early effort on interactive graphics. The graphics are rendered using Vega, a JavaScript library built on the D3 JavaScript library. Vega gallery.

- `vegalite` is an interface to Vega-Lite, a JavaScript library built on top of Vega. Its vignettes and gallery.

- `plotly` is an interface to plotly.js, a charting library built on the D3 library. For details, check out plotly's R site and the book *plotly for R* by Carson Sievert. Normally its graphics rendering requires a 'plot.ly' account. However, according to this site: Plotly's R library "does not require an online plot.ly account."
- `googleVis` is an interface to Google Charts. Here are some examples.
- `highcharter` is an interface to Highcharts, another JavaScript charting library. It is maintained here.

Outside R, or commercial software:

- Tableau is commercial software for visualization. It has gained popularity in recent years.