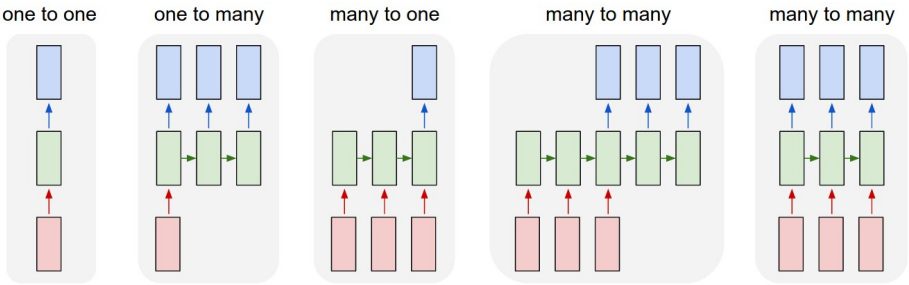
13 PQHS 471 Notes Week 13

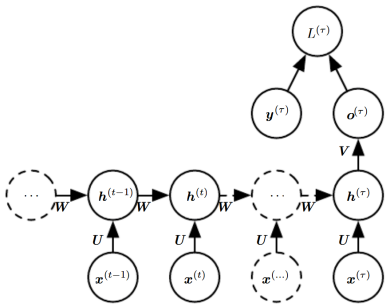
13.1 Week 13 Day 1

13.1.1 Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) are designed to analyze time series data, especially time series predictors. A good description of the various applications of RNNs is in Karpathy's blog. This figure from that blog illustrates various purposes of RNNs.



The "**many-to-one**" type can also be illustrated below (DL Figure 10.5). For this model, an observation consists of a cluster of input vectors and an output (scalar or vector), $\{x^{(1)}, \ldots, x^{(k)}, y\}$, where k may differ across observations. For example, we may want to use the weather information in the past 10 days to predict the temperature tomorrow.



When this RNN model is applied to a cluster of size k, it works in the following way:

- Initialize vector $h^{(0)} = 0$.
- At step t $(t = 1, \ldots, k)$, there is a hidden vector $h^{(t)} = \tanh(Wh^{(t-1)} + Ux^{(t)} + b)$;
- Output $\hat{y} = f(Vh^{(k)} + c)$, where $f()$ is an activation function (e.g., softmax if outcome is multinomial)

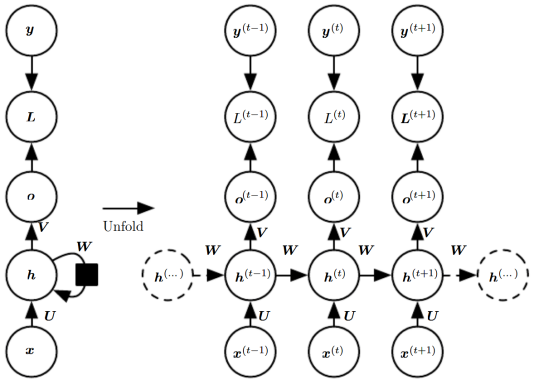
The model could be viewed as a feedforward network with k hidden layers; each hidden layer $h^{(t)}$ has inputs $h^{(t-1)}$ and $x^{(t)}$. But a key difference is that all the layers **share the same weights** W and U and **bias** b. Each hidden layer $h^{(t)}$ outputs to the next layer $h^{(t+1)}$; the last hidden layer outputs to $\hat{y}$, with weight V and bias c. The cost for this cluster is $L(y, \hat{y})$, where $\hat{y}$ is a function of $(x^{(1)}, \ldots, x^{(k)})$. If the input $x^{(t)}$ is p-dimensional and an RNN layer has q units, W is $q \times q$; U is $q \times p$; b is $q \times 1$. So the total number of parameters for the RNN layer is $q(p + q + 1)$.

Because of the parameter sharing, the gradient of $L(y, \hat{y})$ with respect to W is the sum of the gradients with respect to all the individual W's. Similarly for U and b.

Below is an example of Keras code for such an RNN layer. In this example, the number of parameters for the RNN layer is $32 \times (10 + 32 + 1) = 1376$. The 1-dimensional output requires $32 + 1 = 33$ parameters.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1)
summary(model)
```

The "**many-to-many**" type on the right can also be illustrated below (DL Figure 10.3). The model is often shown with a loop as the one on the left; its unfolded version is on the right. For this model, an observation consists of a cluster of input vectors and outputs, $\{x^{(1)}, \ldots, x^{(k)}, y^{(1)}, \ldots, y^{(k)}\}$, where k may differ across observations. For example, an observation can be a sentence, with $x^{(t)}$ a "phrase" of 5 contiguous words and $y^{(t)}$ the next word; a 10-word sentence would form a cluster with $k = 5$.



When this RNN model is applied to a cluster of size k, it works in the following way:

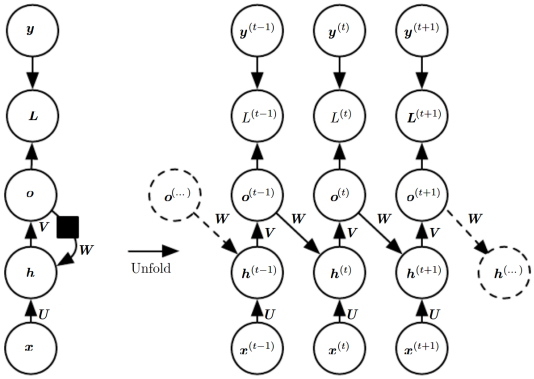
- Initialize vector $h^{(0)} = 0$.
- At step t $(t = 1, \ldots, k)$, there are
  - A hidden vector $h^{(t)} = \tanh(Wh^{(t-1)} + Ux^{(t)} + b)$;
  - An output $\hat{y}^{(t)} = f(Vh^{(t)} + c)$, where $f()$ is an activation function.

Similar to the "many-to-one" model, this model has the **shared weights** W and U and **bias** b. Each hidden layer $h^{(t)}$ outputs to the next layer $h^{(t+1)}$ and to an output $\hat{y}^{(t)}$. The paths to $\hat{y}^{(t)}$ $(t = 1, \ldots, k)$ also **share the same weights** V and **bias** c. The cost for this cluster is $\sum_{t=1}^{k} L(y^{(t)}, \hat{y}^{(t)})$, where $\hat{y}^{(t)}$ is a function of $(x^{(1)}, \ldots, x^{(t)})$. This model has the same number of parameters as the "many-to-one" version, $q(p + q + 1)$.

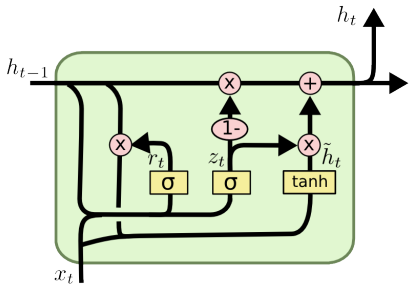
Below is an example of Keras code for such an RNN layer. The specification `return_sequences=T` tells the model the output $h^{(t)}$ at every t; `time_distributed()` tells the model to evaluate the outcome at every t.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 32, return_sequences=T, input_shape = list(NULL, 10)) %>%
  time_distributed(layer_dense(units = 1))
summary(model)
```

An variation of this "many-to-many" model is below (DL Figure 10.4), which differs from the model above in that $h^{(t)} = \tanh(Wo^{(t-1)} + Ux^{(t)} + b)$, where $o^{(t)} = Vh^{(t)} + c$.



GRUs and LSTMs: Some RNNs have sophisticated operations inside a unit. One example is the **gated recurrent unit** (GRU) (Cho et al., 2014). The figure below is a "fully gated" version (from Olah's blog).



Below is the matrix representation of the operations across all units:

$$\begin{cases} r_t &= \sigma(W_r x_t + R_r h_{t-1} + b_r), \\ z_t &= \sigma(W_z x_t + R_z h_{t-1} + b_z), \\ \tilde{h}_t &= \tanh(W_a x_t + R_a(r_t \circ h_{t-1}) + b_a), \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t. \end{cases} \quad \text{update gate}$$

Here the operator $\circ$ is the Hadamard product (i.e., element-wise product between two vectors). For a single unit, r_t, z_t, h_t are scalars, and $\circ$ becomes a regular product. If the input $x^{(t)}$ is p-dimensional and an RNN layer has q GRUs, W_r, W_z, and W_a are $q \times p$; R_r, R_z, and R_a are $q \times q$; b_r, b_z, and b_a are $q \times 1$. So the total number of parameters is $3q(p + q + 1)$.

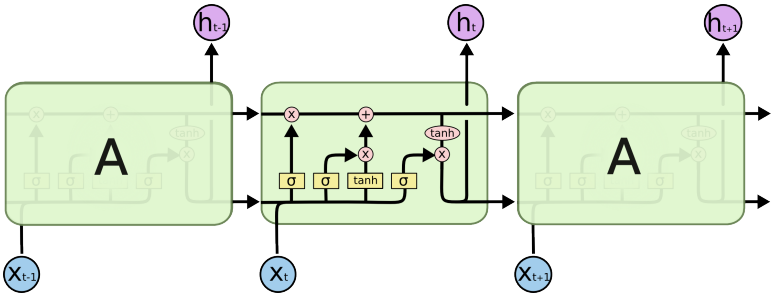
Below is an example of Keras code for a layer of GRUs. In this example, the number of parameters for the GRU layer is $3 \times 32 \times (10 + 32 + 1) = 4128$. The 1-dimensional output requires $32 + 1 = 33$ parameters.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1)
summary(model)
```

One can stack a GRU layer after another by specifying `return_sequences=T` in the GRU layer feeding to the next. The second GRU layer treats the $h^{(t)}$ from the first GRU layer as its $x^{(t)}$. In the example below, the second GRU layer has 64 units, requiring $3 \times 64 \times (32 + 64 + 1) = 18,624$ parameters.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, return_sequences=T, input_shape = list(NULL, 10)) %>%
  layer_gru(units = 64, activation = "relu") %>%
  layer_dense(units = 1)
summary(model)
```

GRUs is a simplified version of the **long short-term memory** (LSTM) network. An LSTM unit looks like the figure below. It has a separate "memory" thread in addition to the "hidden" thread. A nice description of the LSTM is given by Christopher Olah. An LSTM layer with q units has $4q(p + q + 1)$ parameters.



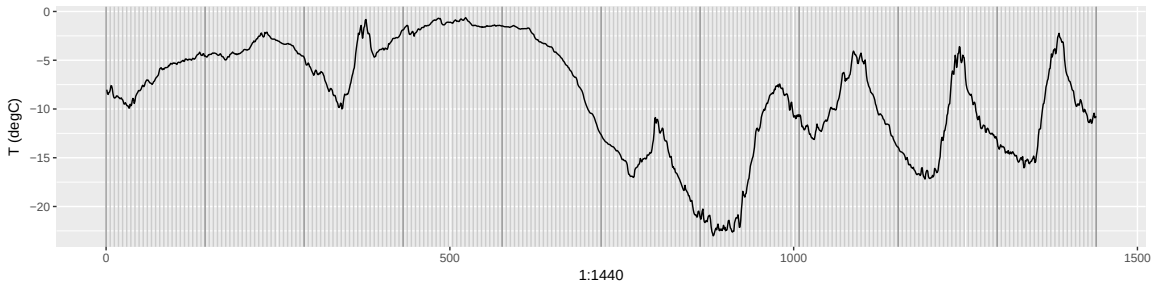

```
library(keras)
model <- keras_model_sequential() %>%
  layer_lstm(units = 32, input_shape = list(NULL, 10)) %>%
```

```
  layer_dense(units = 1)
summary(model)
```

13.1.2 Example: Time series forecasting

https://tensorflow.rstudio.com/blog/time-series-forecasting-with-recurrent-neural-networks.html

In this weather time series dataset, temperature and 13 other relevant variables (e.g., air pressure) are recorded every 10 minutes for 8 years from 2009 to 2016 in Jena, Germany. Below are the temperatures in the first 10 days of 2009.



The authors want to predict the temperature 24 hours from now given the data in the last 10 days (temperature + 13 other variables). Instead of using the full 10 days data as the input, they sampled hourly (i.e., took every 6th observation; $k = 240$). In a FFNN, there would be a total of $240 \times 14 = 3360$ input neurons. For the RNN model below, the GRU layer would have 240 unfolded layers, each with a 14-vector as the input, and a scalar output after the last unfolded layer.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, 14)) %>%
  layer_dense(units = 1)
summary(model)
```

13.1.3 Doughnut experiment

See the separate .Rmd file.

13.1.4 Assignment

1. Reading for next lecture: ISLR 10.3

13.2 Week 13 Day 2

In **supervised learning**, the data always have two parts, X and Y, and we focus on learning how to use X to make inference about Y. The outcome Y drives the learning process. In machine learning, inference has been mostly about prediction.

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

13.2.1 ISLR 10.3 Clustering

Clustering is to divide data into subsets of relatively homogeneous observations (i.e., a partition). Clustering shares some similarity with:

- Trees, which give an outcome-driven, cookie-cutter partitioning of X;
- Kernel SVMs, which use pairwise similarity measures on X.

13.2.2 K-means clustering

For the K-means method, we need (1) a measure of **dissimilarity** between any two points, (2) a definition of **centroid** for any set of data points, and (3) a number K.

Goal: Given K, find the partition $\{C_1, \ldots, C_K\}$ of data indices that minimizes the total within-cluster "heterogeneity". That is,

$$\text{minimize}_{\{C_1,\ldots,C_K\}} \sum_{k=1}^{K} W(C_k), \text{ where } W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} d(x_i, x_{i'}). \tag{10.9}$$

Here $d(x_i, x_{i'})$ is a measure of **dissimilarity** between x_i and $x_{i'}$, and $W(C_k)$ is a measure of heterogeneity within cluster C_k.

A popular choice for $d()$ is the **squared Euclidean distance** $d(x_i, x_{i'}) = \|x_i - x_{i'}\|^2 = \sum_j (x_{ij} - x_{i'j})^2$. Often 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 $\frac{1}{|C_k|} \sum_{i \in C_k} x_i$. It is the point that minimizes the total squared Euclidean distance from all the observations in the cluster.

K-means algorithm is a greedy algorithm. A generic version of the algorithm is below (shown in 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., 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:

- 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. 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). There is 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 (proof below)

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

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

Note: It is helpful to distinguish these related concepts: (1) a goal, (2) an algorithm, (3) an implementation of the algorithm (which is often called an algorithm itself). For example,

(1) Our goal is to identify the partition that minimizes (10.9);
(2) The algorithm above is a way (which is easy to understand) to achieve or approximate our goal;
(3) The software implementation is a faster way that is equivalent to (2) (but may not be as easy to understand).

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 ridge criterion with $\lambda = 10$ (alternative criteria: least squares, lasso);
(3) We use gradient descent to fit the model (alternative algorithms: Newton–Raphson, direct formula, etc.)

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).

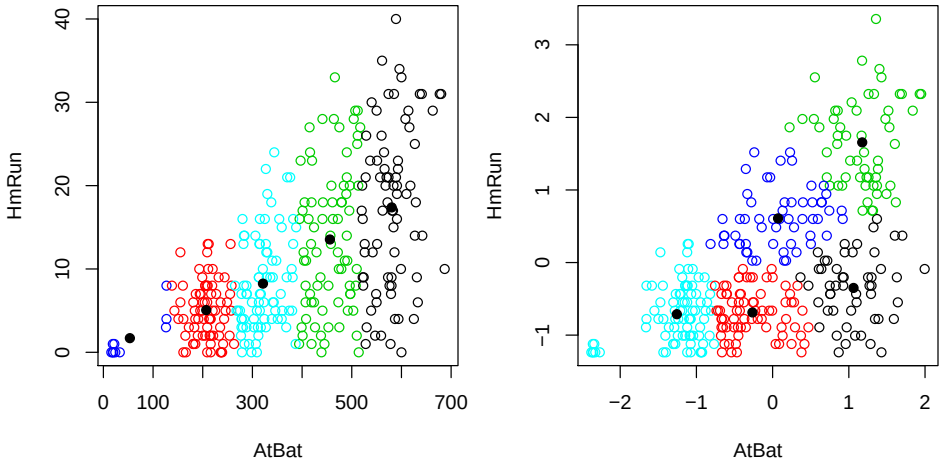
The base R has a function `kmeans()` for performing the K-means clustering using squared Euclidean distance. Unfortunately it does not even have the option for scaling the features!

```
library(ISLR)
Hitters1 = Hitters[,1:7]
names(Hitters1); dim(Hitters1)
aa = kmeans(Hitters1, 3, nstart=10)
str(aa)

aa = kmeans(Hitters1, 3, nstart=1); aa$tot.withinss  ## repeat to see the effect of nstart

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

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

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

```
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)
```



13.2.3 Hierarchical clustering

For the hierarchical clustering method, we need (1) a measure of **dissimilarity** between any two subsets, and (2) a number K or a **threshold** value for dissimilarity to determine if two subsets belong to the same cluster. (1) may not be easily defined directly on all subsets. It is often replaced by (1a) defining a measure of **dissimilarity** between data points and (1b) a way (called **linkage**) to define (1) using (1a).

Hierarchical clustering is a *bottom-up* approach. In contrast, trees are built *top-down*. It requires a measure of **dissimilarity**, $d(C_i, C_j)$, between any two non-overlapping subsets of observations, C_i and C_j. The smaller $d(C_i, C_j)$ the more similarity between C_i and C_j. The algorithm is:

1. Start from n singletons (a singleton is a subset containing a single observation).
2. Grow a tree by repeating this until we reach a single set: Among all the current subsets, merge the two subsets that have the smallest $d(C_i, C_j)$, and record that $d(C_i, C_j)$.
3. Use a cutoff value for $d(C_i, C_j)$ to cut the tree to obtain branches as clusters (as shown in ISLR Figure 10.9).

Notes:

- 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 cutoff value is a hyperparameter.
- ISLR Figures 10.10 and 10.11 illustrate this algorithm.

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 singletons, $d(x_i, x_j)$. Then $d(C_i, C_j)$ can be defined through $d(x_i, x_j)$, in one of the following ways:

- $d(C_1, C_2) = \max_{i \in C_1, j \in C_2} d(x_i, x_j)$ (**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)

Notes:

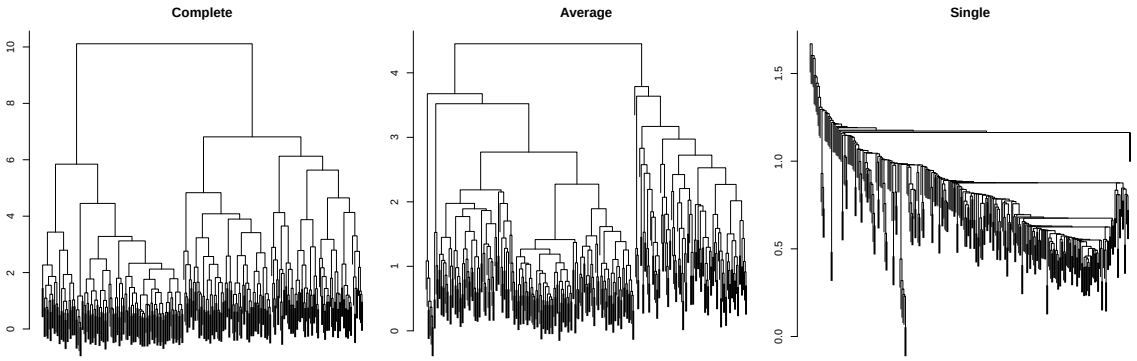
- Examples in ISLR Figure 10.12; also see below.
- Single linkage often has the signature pattern of adding one observation at a time.
- Common choices for dissimilarity $d(x_i, x_j)$ are:
  (1) Euclidean distance (often requiring **feature standardization** to make sense);

(2) One minus correlation coefficient (Pearson or Spearman or other measures). The correlation is between two observations across features, and so **feature standardization** is often necessary for the results to make sense.
(3) If the data are binary indicating presence or absence of features, one minus Jaccard Index is often used. The Jaccard index between two observations is (# features present in both) / (# features present in either). (In R package `vegan`, `vegdist()` calculates various dissimilarity indices including one minus Jaccard.)

- Different measures of dissimilarity can reflect quite differently on whether two observations are "similar" or not. ISLR Figure 10.13 shows an example for (1) and (2).

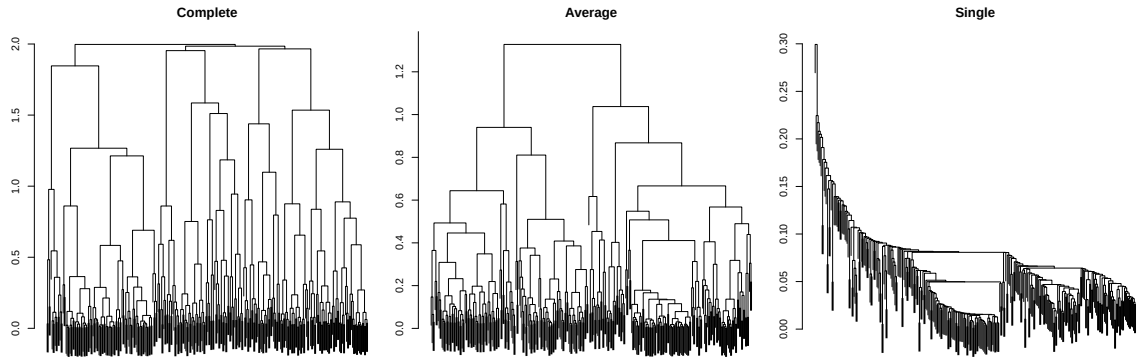
The base R has `hclust()` for hierarchical clustering, which by default uses complete linkage. It takes a `dist` object, which can be generated by calling `dist()` or `as.dist()`. `cutree()` cuts a `hclust` tree into clusters. The function `dist()` by default computes the Euclidean distance between every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

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

cluster1 = cutree(hclust(dist1), 5)
cluster2 = cutree(hclust(dist1), h=5.7)
table(cluster1); table(cluster2)
table(cluster1, cluster2)  ## same clustering

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")
```



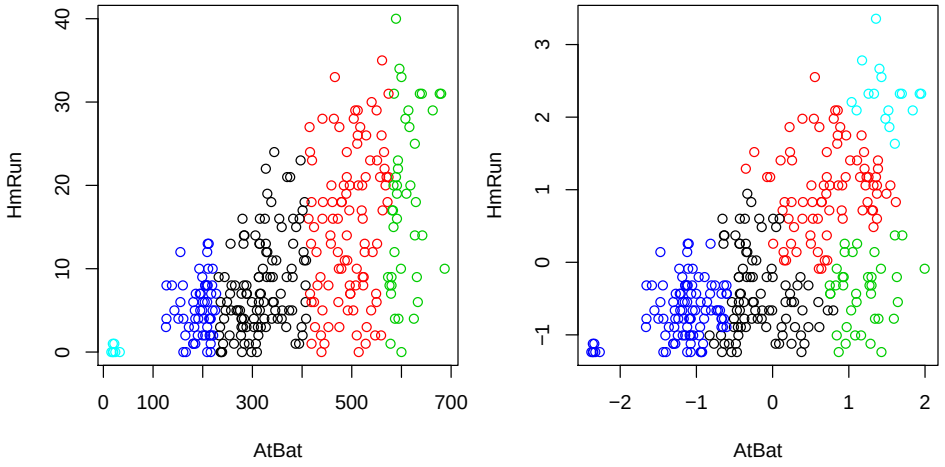
Now we try the correlation approach.

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

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")
```



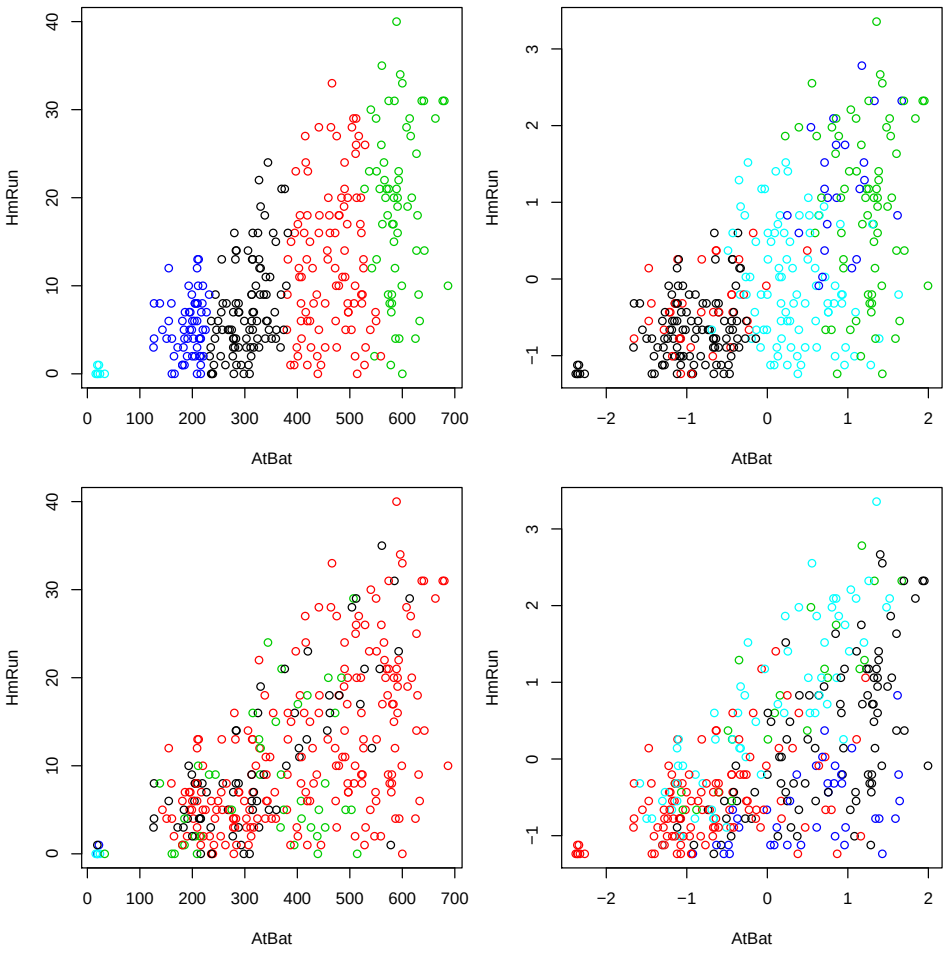
We now use two features to demonstrate the difference between scaling and not scaling features.

```
Hitters3 = Hitters1[,c(1,3)]
Hitters4 = scale(Hitters3)

aa = cutree(hclust(dist(Hitters3)), 5)
bb = cutree(hclust(dist(Hitters4)), 5)

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.

```
aa = cutree(hclust(dist(Hitters1)), 5)
bb = cutree(hclust(dist(Hitters2)), 5)
cc = cutree(hclust(as.dist(1-cor(t(Hitters1)))), 5)
dd = cutree(hclust(as.dist(1-cor(t(Hitters2)))), 5)
table(aa, cc); table(bb, dd)

## Plot the results with 2 features as backdrop
plot(Hitters3, col=aa); plot(Hitters4, col=bb)
plot(Hitters3, col=cc); plot(Hitters4, col=dd)
```



13.2.4 Clustering using mixture models

For the mixture model approach, we need (1) a family of distributions (with parameters θ), and (2) a number K.

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:

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

where $\phi = (\pi_1, \ldots, \pi_K, \theta_1, \ldots, \theta_K)$. Once we have estimated the parameters, we can calculate the posterior probability for each observation as

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

and assign the observation to the class with the largest posterior probability $\hat{p}_k$.

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, and has known classes and often known probabilities π_k for the observations.

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,

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

When there are no x (or a constant x across all observations), **this 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.]

13.2.5 Example: Clonal detection with tumor 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)

13.2.6 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

13.2.7 Assignment

1. Reading for next lecture: ESL 14.2