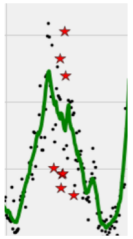
Clustering

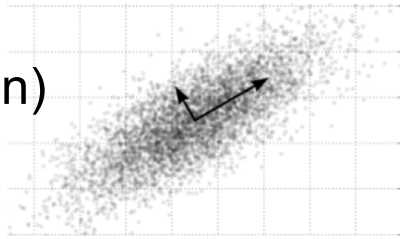
Unsupervised learning

Terence Parr
MSDS program
University of San Francisco

Unsupervised learning overview

- In a nutshell, we have just X not $X \rightarrow y$ and would like to know about X, such as density or interesting subregions/vectors of X (n x p matrix)
- Clustering (unsupervised classifier; i.e., no known classes)
  - k-means / k-medioid / mean-shift
  - hierarchical clustering
  - spectral clustering (graph connecting observations (nodes) by distance-labeled edges)
- Recommendation engines
  - collaborative filtering ("other people like you bought X")
  - market basket analysis / association rules ("what do people buy together?); see *a priori algorithm*
- Principle components analysis (orthogonal vectors of most variation)
- Page rank for ranking most important articles/nodes
- Word embeddings like glove (matrix factorization)
- Anomaly detection (fraud or network attack detection)





See Chapter 14 in ESLII

The problem is…

- All of those unsupervised methods can be useful
- But a big problem is that there no clear measure of success, such as the metrics used by supervised learning
- E.g., you have bank transactions and no idea which are fraudulent; design algorithm to identify fraud; now, how do you know if your algorithm works?
- You can measure cluster centroid separation but it still doesn't truly indicate proper clustering; might have too many k etc…

"Almost all of AI's recent progress is through one type, in which some input data (A) is used to quickly generate some simple response (B)."
Andrew Ng in *What Artificial Intelligence Can and Can't Do Right Now*
Harvard Business Review November 9, 2016

Warning...

- Clustering sounds awesome but useful only in limited circumstances, such as vector quantization & compression, and usually only when p is small
- Also, clustering generally doesn't work well with imbalanced data sets such as fraud or network attack classification

Is clustering really what we want anyway?

- Imagine clustering a customer db into 4 clusters; now what?
- You have 4 groups of, say, 4 million records each; what do you do with it?
- Let's say you can identify marketing-related groups like "technerd", "shoeshopper", etc… What do you do with that info?
- Old joke: You know you're wasting half of your marketing money, but you don't know which half!
- Can try to market to those groups but don't we really want to know what kind of ad people click on? Run an ad campaign and track customer->clicks; now you have a supervised problem

Can also try semi-supervised learning

- Sometimes getting labels is expensive or difficult, such as in medicine
- Use a few labeled observations to get things started, such as picking the initial centroids (cluster centers)
- You can try to get your client to give you class labels for a few observations
- Still, it's good idea to try to turn unsupervised into supervised learn problem so let's try to start with this "kernel" and gradually broaden the labeled data
- For a good summary, see **An overview of proxy-label approaches for semi-supervised learning** by S. Ruder https://ruder.io/semi-supervised/index.html#selftraining

Self-training procedure

1. Get small initial X^0, y^0 training set from X, y
2. Train supervised model M^0 on initial X^0, y^0 set
3. Use model M^0 to make predictions for $X - X^0, y - y^0$
4. Combine highest confidence predictions with X^0, y^0 to get, new larger labeled set X^1, y^1
5. Train model M^1 on X^1, y^1
6. Repeat until all X, y are labeled or no high confidence obs.

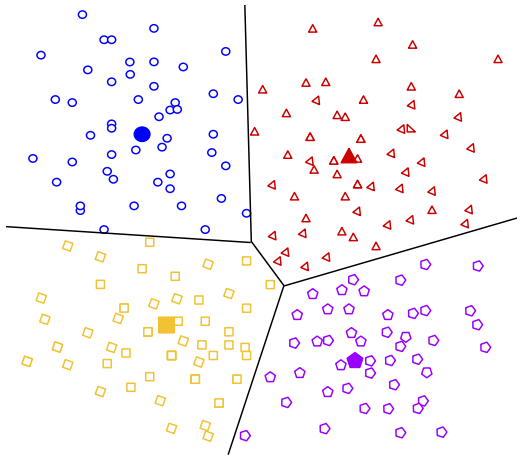
Selecting "highest confidence" metric requires experimentation and requires accurate confidence or probabilities from the model

E.g., see **Semi-Supervised Self-Training of Object Detection Models**
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.3602&rep=rep1&type=pdf

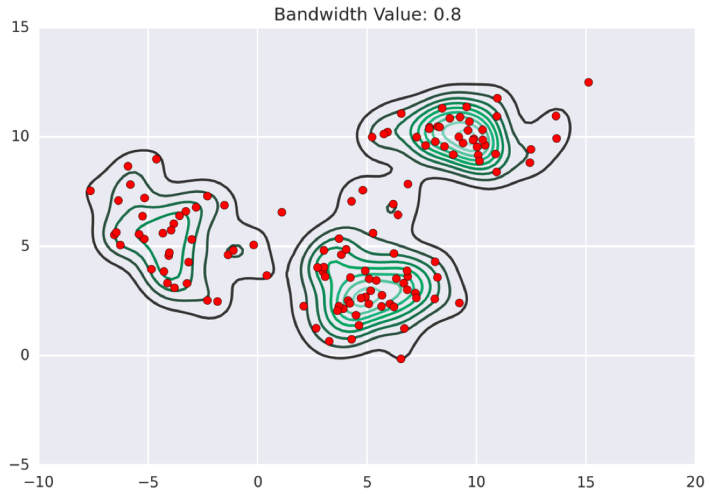
Clustering preliminaries

- Each $x^{(i)}$ in X is a point in p space with p coordinates
- Space can be Euclidean but categorical vars present challenges
- All such spaces must have distance($x^{(i)}$, $x^{(j)}$) measure
- Often we need to normalize x values so distance measures same thing in all directions
- L_1 and L_2 are common distances for Euclidean space
- L_∞ also useful: max abs difference in any dimension
- For large p and/or binary values, better to use cosine similarity (angle between 2 vectors); $\cos(\theta)=\frac{v \cdot w}{\|v\|\|w\|}$ so 1-$\cos(\theta)$ is distance
- Two flavors: point-assignment and agglomerative

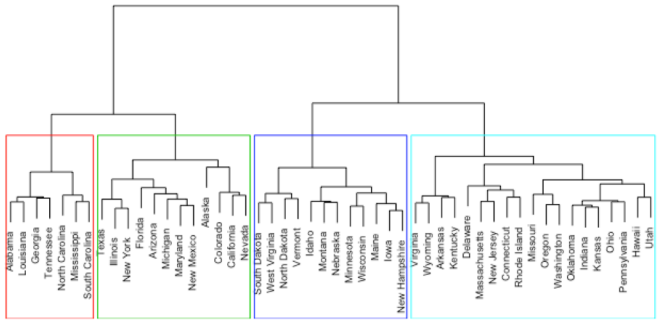
Clustering examples



k-means



mean-shift

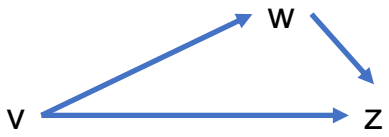


hierarchical clustering

Images: https://developers.google.com/machine-learning/clustering/clustering-algorithms, https://spin.atomicobject.com/2015/05/26/mean-shift-clustering/, https://uc-r.github.io/hc_clustering

Distance measure requirements

1. Always nonnegative; only distance(v,v) is 0
2. Symmetry; distance(v,w) = distance(w,v)
3. Triangle inequality; distance(v,w)+distance(w,z)≥distance(v,z)

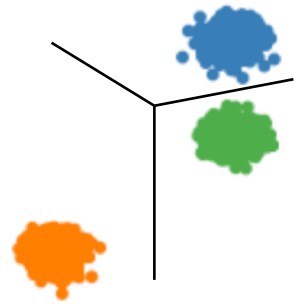


From http://www.mmds.org/; see it for more on distance metrics

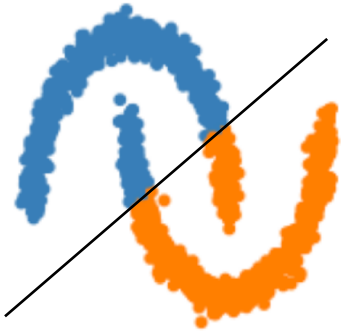
k-means clustering

- Assumes Euclidean space
- Clusters separated by straight lines only
- User provides X and number of clusters to find, k
- Idea is to pick k centroids in p space and assign points to cluster with closest centroid then recompute centroids
- Repeat until the cluster assignments stop changing
- Can select k points as initial centroids:
  - At random (seems to do a crappy job for large p)
  - By picking k distant points (*k-means*++ is a variation for initial selection)
- Algorithm converges using Euclidean distance
- Not guaranteed to find optimal clusters

k-means works

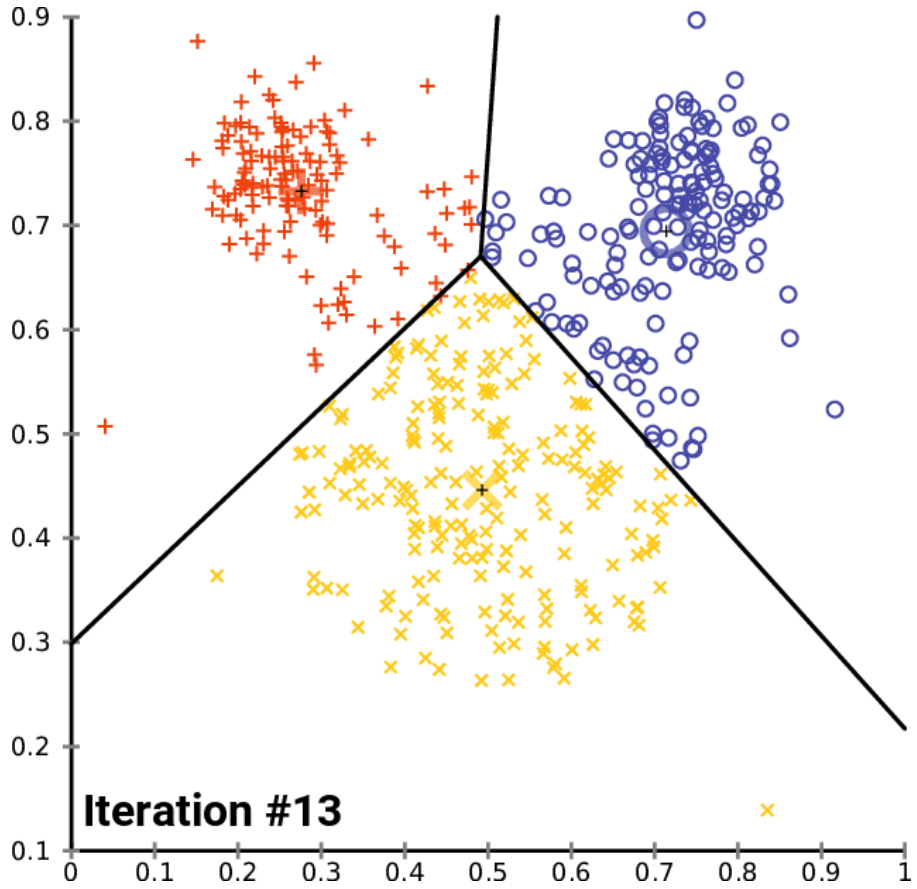


k-means fails



k-mean clustering animation

- k-means gives Voronoi tessellation
- It assumes that all points in the cluster are contiguous; sounds obvious, but for most real problems this assumption doesn't hold
- If true clusters are noncontiguous, k-means will give poor results



Animation from https://en.wikipedia.org/wiki/K-means_clustering

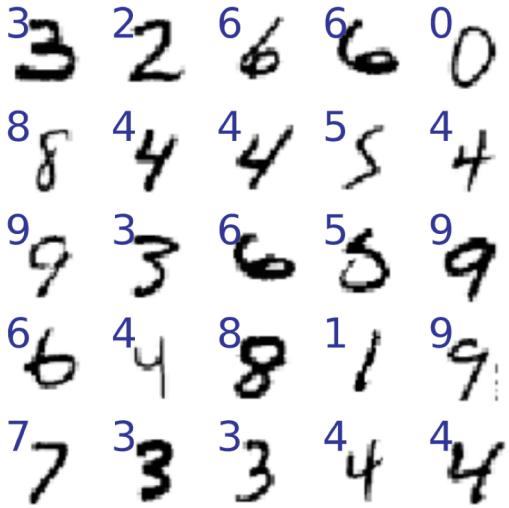
k-means algorithm

Algorithm: *kmeans(X,k)*

Select k points from X as initial centroids $m_{1..k}^{(t=0)}$ for clusters $C_{1..k}^{(t=0)}$
repeat
  foreach $x \in X$ **do**
    $i = \arg\min_j distance(x, m_j^{(t)})$ (*find closest centroid to x*)
    Add x to cluster $C_i^{(t+1)}$ (*assign x to cluster*)
    for $i = 1..k$ **do**
      $m_i^{(t+1)} = \frac{1}{|C_i^{(t+1)}|} \sum_{x \in C_i^{(t+1)}} x$ (*recompute centroids*)
    end
  end
  $t = t + 1$
until $C_{1..k}^{(t)} = C_{1..k}^{(t-1)}$ (*until clusters don't change*)

k-means application: MNIST

(Known digits so we can measure error)



- Goal: cluster MNIST digit greyscale images
- p=28x28=784 pixels/image
- Results vary a lot but close to perfect score depending on initial cluster centroid selection
- Somehow clusters are pretty contiguous, so k-means works ok

See https://github.com/parrt/msds621/blob/master/notebooks/clustering/kmeans.ipynb

Try different subsets, initial centroids

```
results = []
for i in range(10):
    n = 1000
    df_subset = df_digits.sample(n=n, replace=False)
    X = df_subset.drop('digit', axis=1)
    X = normalize(X)
    y = df_subset['digit']
    kmeans = KMeans(10, init='random')
    kmeans.fit(X.values)
    y_pred = kmeans.predict(X)
    correct = np.sum(y!=y_pred)
    print(f"{correct}/{n}={100*correct/n:.1f}% correct")
    results.append(correct)
print(f"Avg {np.mean(results)}/{n}=…")
```

```
866/1000=86.6% correct
991/1000=99.1% correct
899/1000=89.9% correct
993/1000=99.3% correct
909/1000=90.9% correct
746/1000=74.6% correct
917/1000=91.7% correct
904/1000=90.4% correct
989/1000=98.9% correct
860/1000=86.0% correct
Avg 907.4/1000=86.0% correct
stddev 7.15
```

k-means application: Breast cancer

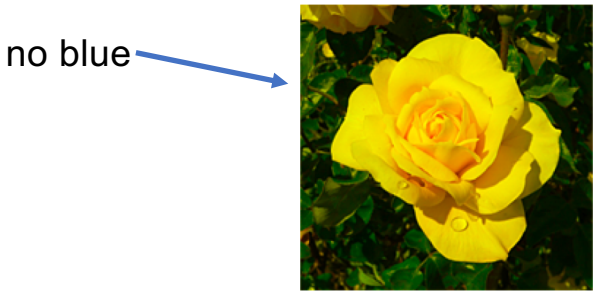
(Known cancer/benign target so we can measure error)

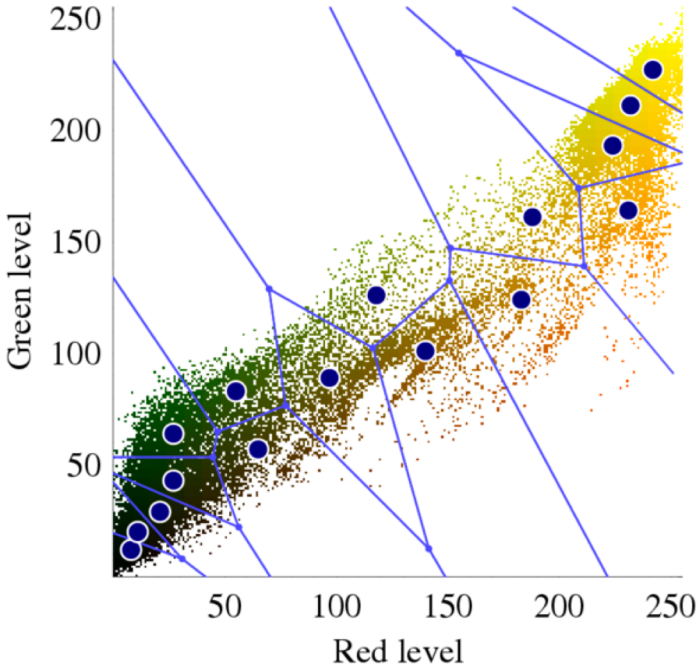
- k-means (even using k-means++ centroid selector) isn't great
- Does about 11-89% accuracy, huge stddev; i.e., doesn't work very well on this data set
- Dimensions are p=30, which could be part of the issue but most likely true clusters have noncontiguous regions; k-means will fail in this situation

505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
505/569=88.8% correct
64/569=11.2% correct
505/569=88.8% correct
64/569=11.2% correct
505/569=88.8% correct
64/569=11.2% correct
64/569=11.2% correct
505/569=88.8% correct
Avg 387/569=68.1% correct
stddev 34.27

k-means application: color quantization

- Color pictures typically use lots of unique colors, possibly 10s of thousands
- Each pixel in the image has Red/Green/Blue colors, 1 byte per RGB = 3 bytes (24 bits)
- Each RGB is a 3D coordinate of color: (R, G, B), with possibly tens of thousands of unique combinations
- Example with just red/green (omit blue):

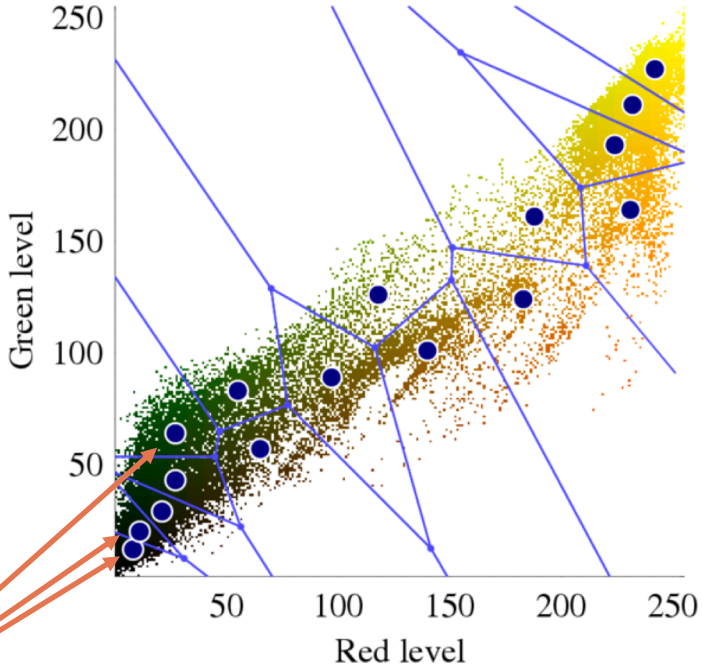




See https://scikit-learn.org/stable/auto_examples/cluster/plot_color_quantization.html
Image from https://en.wikipedia.org/wiki/Color_quantization

Color quantization cont'd

- (R,G,B) takes 3 bytes per pixel which makes images really big
- If a picture only has 256 unique colors we can map all (R,G,B) vectors to a single byte; the color "index" 0..255 points into a color palette with the full (R,G,B) vectors; 3x compression for each pixel so a massive compression
- If picture has more than 256 colors, we can cluster in RGB space with k=256 and it will group similar colors together; then we pick the centroid as the colors in the palette



Color quantization example, k=10

Original image
(96,615 colors)

Quantized image
(10 colors, k-Means)

Quantized image
(10 colors, at random)

See https://github.com/parrt/msds621/blob/master/notebooks/clustering/kmeans.ipynb

Color quantization example, k=4

Original image
(96,615 colors)

Quantized image
(4 colors, k-Means)

Quantized image
(4 colors, at random)

Confusion point

- k-means' centroids don't have to be points in X, usually aren't
- *k-medians* uses median not mean for centroids and, thus, minimizes w.r.t. L1 not L2 distance; median even for single dimension doesn't have to be point in $x^{(i)}$ space
- *k-medoids* (not spelled *k-medioids*) requires medoids to be points in X; works with any distance measure; sounds like k-means but algorithm is pretty different; gotta pick "centrally located point"

Trouble with k-means

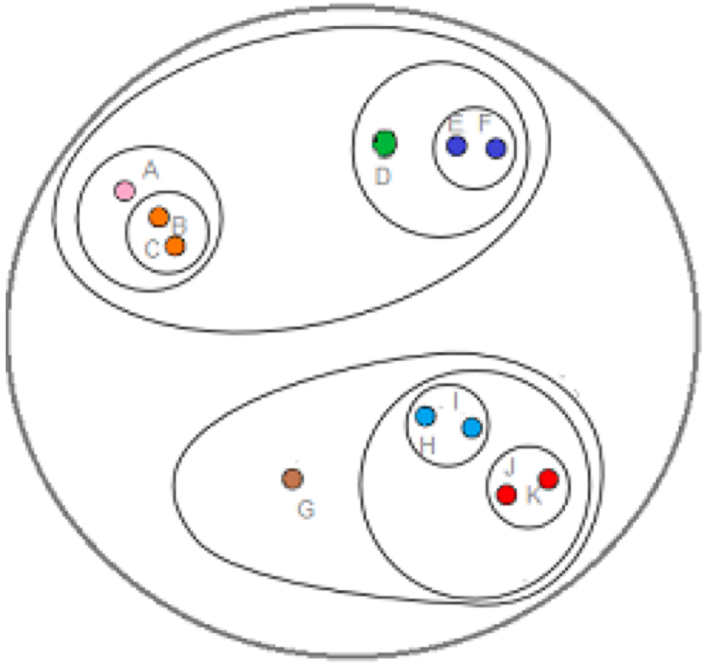
- k-means requires user give number of clusters k
- Picking k is usually a problem
- Color quantization and MNIST digits have known k, but few do
- Different starting centroids can lead to different results
- Each obs. is forced into one of the k clusters, but probabilities might be nice; we could use distance to centroid I guess but a density estimate would be better

Hierarchical (agglomerative) clustering

- **Idea**: put every point into its own singleton cluster; repeatedly group two closest clusters into a meta-cluster until there is a single cluster left
- Can also stop when distance between clusters sufficiently large
- We need a cluster distance metric; called the *linkage criterion*
- Simplest linkage is just the distance between cluster centroids
- Result is a tree of clusters, one cluster per level
- We get all possible clusters up to k=n-1 for n observations

Dendograms

- Dendogram is a tree of clusters, one cluster per level
- Distance from node to children reflects between-cluster-metric



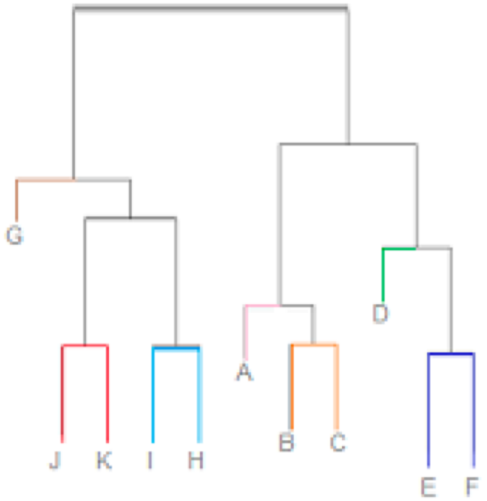
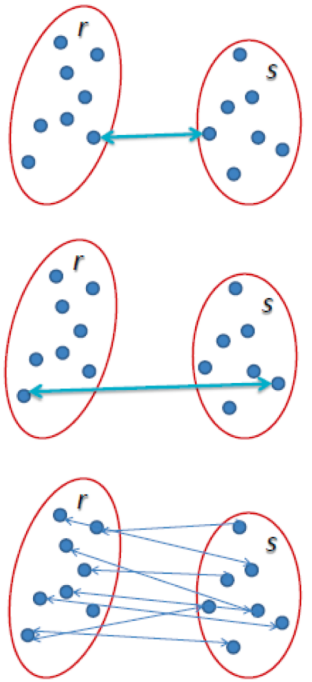


Image from https://www.statisticshowto.datasciencecentral.com/hierarchical-clustering/

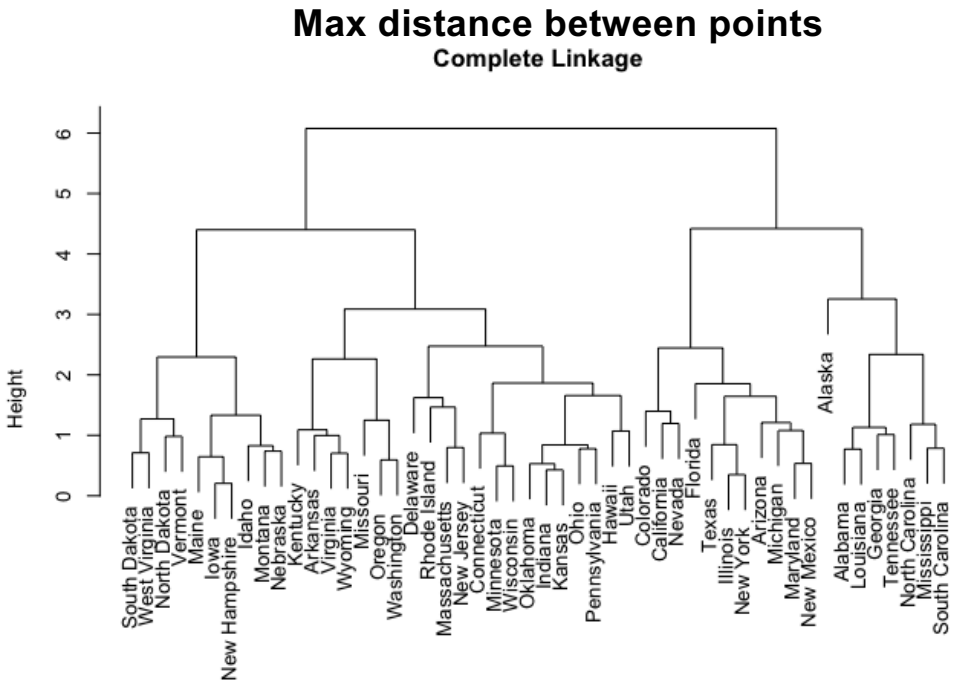
Between-cluster distance metrics (*linkage*)

1. Minimum distance between any two points in the cluster (*single linkage*); tends not to get compact clusters and can get chains of points
2. Max distance between point pairs (*complete linkage*); tends to get compact clusters but points can be closer to other clusters than those within their cluster
3. Average distance of all point pairs from two clusters (*group average linkage*); tries to get compact clusters that are far apart
4. *Ward's method* minimizes within-class variance; merge pair with smallest variance at each step



Images from http://saedsayad.com/clustering_hierarchical.htm

Effect of between-cluster-metric



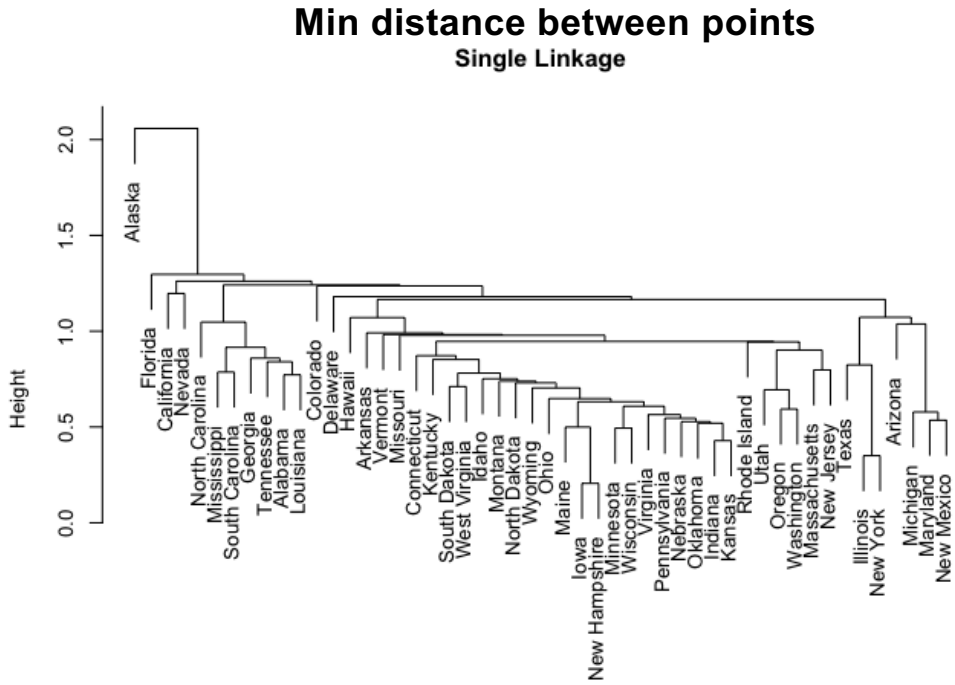


Image from https://uc-r.github.io/hc_clustering

UNIVERSITY OF SAN FRANCISCO

Ward's method

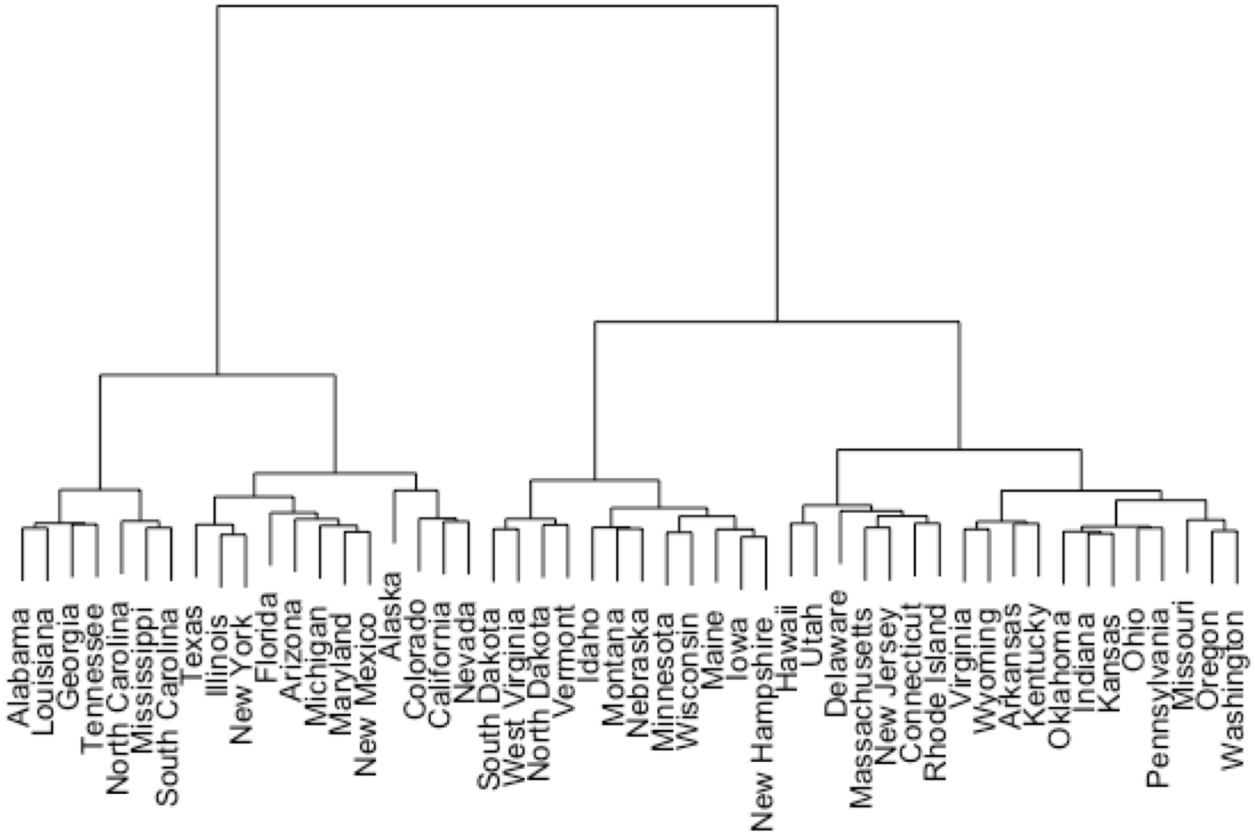


Image from https://uc-r.github.io/hc_clustering

Non-numeric clustering

- How do you cluster documents?
- Can use edit distance or Jaccard similarity between text docs
- Maybe convert words to Glove word vectors
- But what about tabular data with nominal categorical variables?
- In non-numeric space, what is a centroid vector?
- There are similarity measures for categoricals, but I'm not a big fan, particularly with mixed numeric and categorical data

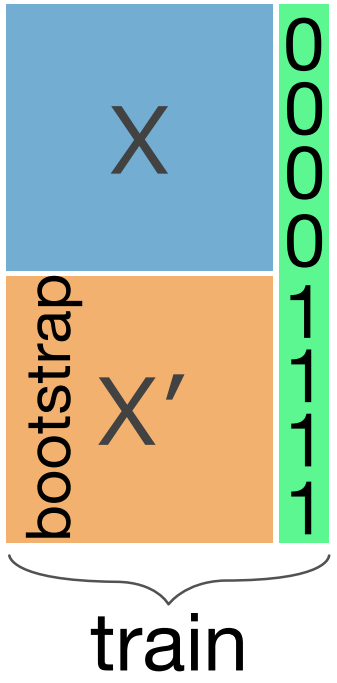
Breiman's RF clustering

- Goal: similarity($x^{(i)}, x^{(j)}$) or distance($x^{(i)}, x^{(j)}$) for any two feature vectors in X, even in the presence of mixed categorical and numeric data
- Random Forests to the rescue again with clever trick that turns unsupervised into the supervised problem then drives similarity matrix between all $x^{(i)}, x^{(j)}$ pairs
- Proximity matrix: count how often $x^{(i)}, x^{(j)}$ appear in same leaf in all trees of forest; normalize by number of leaves
- Use 1 minus proximity to get distance, then can use any clustering algorithm we want like k-means, …

See https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Random Forest distance metric

1. Consider all X records as as class 0
2. Duplicate and bootstrap columns of X to get X': class 1
   - Breiman: X' created by "…*sampling at random from the univariate distributions…*" of X
   - X' destroys relationships between columns of X
3. Create y to label X vs X'
4. Train RF on stacked [X,X'] → y
5. Walk all leaves of all trees, bumping proximity[i,j] for all $x^{(i)}, x^{(j)}$ pairs in leaf; divide proximities by num of leaves
6. Cluster using 1-proximity for distance matrix



Breiman's RF distance X' from X

Here's how to create X' from X

```
def df_scramble(X : pd.DataFrame) -> pd.DataFrame:
    X_rand = X.copy()
    for colname in X:
        X_rand[colname] = \
            np.random.choice(X[colname].unique(),
                             len(X),
                             replace=True)
    return X_rand
```

Breiman's RF distance conjure supervised from unsupervised

```
def conjure_twoclass(X : pd.DataFrame)\
                    -> (pd.DataFrame, pd.Series):
    X_rand = df_scramble(X)
    X_synth = pd.concat([X, X_rand], axis=0)
    y_synth = np.concatenate([np.zeros(len(X)),
                              np.ones(len(X_rand))],
                             axis=0)
    return X_synth, pd.Series(y_synth)
```

Trains model to recognize structure between variables, but goal is simply co-existence in leaves