Unsupervised learning

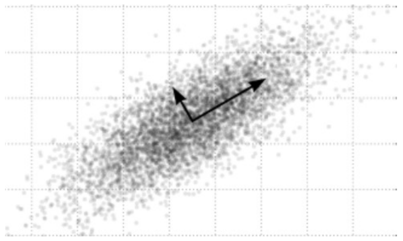
Mostly clustering

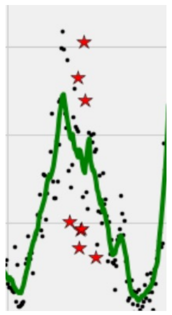
Terence Parr
MSDS program
University of San Francisco

See Chapter 14 in ESLII

Unsupervised learning techniques…

- 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)
- Principle components analysis (orthogonal vectors of most variation)
- Page rank for ranking most important articles/nodes
- Collaborative filtering (recommending movies)
- Anomaly detection (fraud or network attack detection)





"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

UNIVERSITY OF SAN FRANCISCO

Most common unsupervised learning

- Clustering (unsupervised classifier; i.e., no known classes)
  - k-means / k-medoid / mean-shift
  - hierarchical clustering
  - spectral clustering; graph connecting observations (nodes) by distance-labeled edges
- Recommendation engines (this stuff works great)
  - collaborative filtering ("other people like you bought X"); best done with embeddings; e.g., see [1]
  - market basket analysis / association rules; see *a priori algorithm* ("what do people buy together?)

[1] https://github.com/fastai/fastbook/blob/master/08_collab.ipynb

The problem with clustering is…

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

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; you just 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

Instead can try semi-supervised learning

- Sometimes getting labels is expensive or difficult, such as in medicine
- Still, it's good idea to try to turn unsupervised into supervised learning problem so let's try to start with this "kernel" and gradually broaden the labeled data
- Use a few labeled observations to get things started, such as picking the initial centroids (cluster centers); try to get your client to give you class labels for a few observations
- 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

One possible self-training procedure

1. Get small initial X^0,y^0 labeled training set from X
2. Train supervised model M^0 on initial X^0,y^0 set
3. Use model M^0 to make predictions for $X \backslash X^0$
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 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

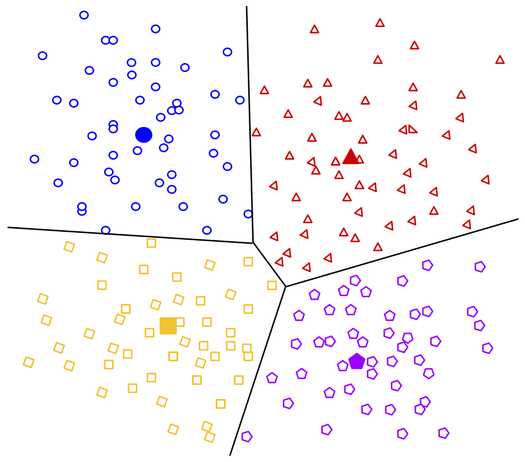
Clustering

UNIVERSITY OF SAN FRANCISCO

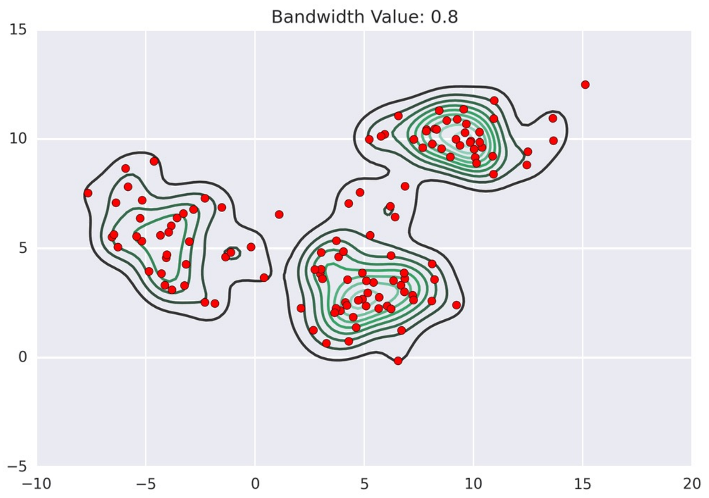
Clustering preliminaries

- Each $x^{(i)}$ in X is a point in p space with p coordinates
- Space can be Euclidean but categorical vars present a challenge
- All such spaces must have $\underline{\text{distance}(x^{(i)}, x^{(j)})}$ measure
- Often we need to normalize x values so distance means 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(θ)=$\frac{v \cdot w}{\|v\|\|w\|}$ so 1-cos(θ) is distance
- Two flavors: point-assignment and agglomerative

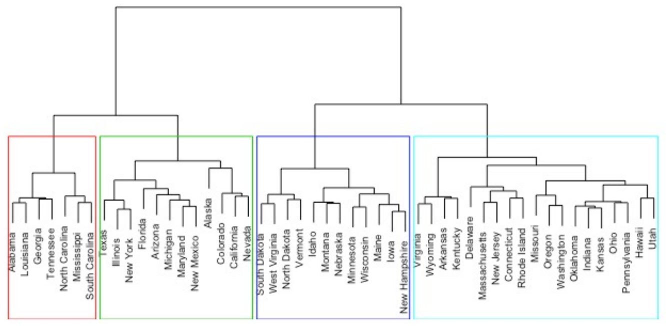
Clustering examples



k-means



mean-shift



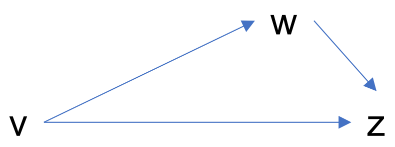
hierarchical clustering

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

UNIVERSITY OF SAN FRANCISCO

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)

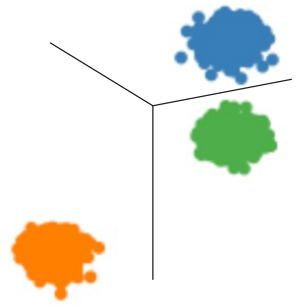


From http://www.mmds.org/; see it for more on distance metrics

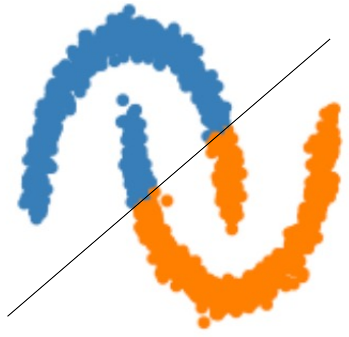
k-means clustering

- 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



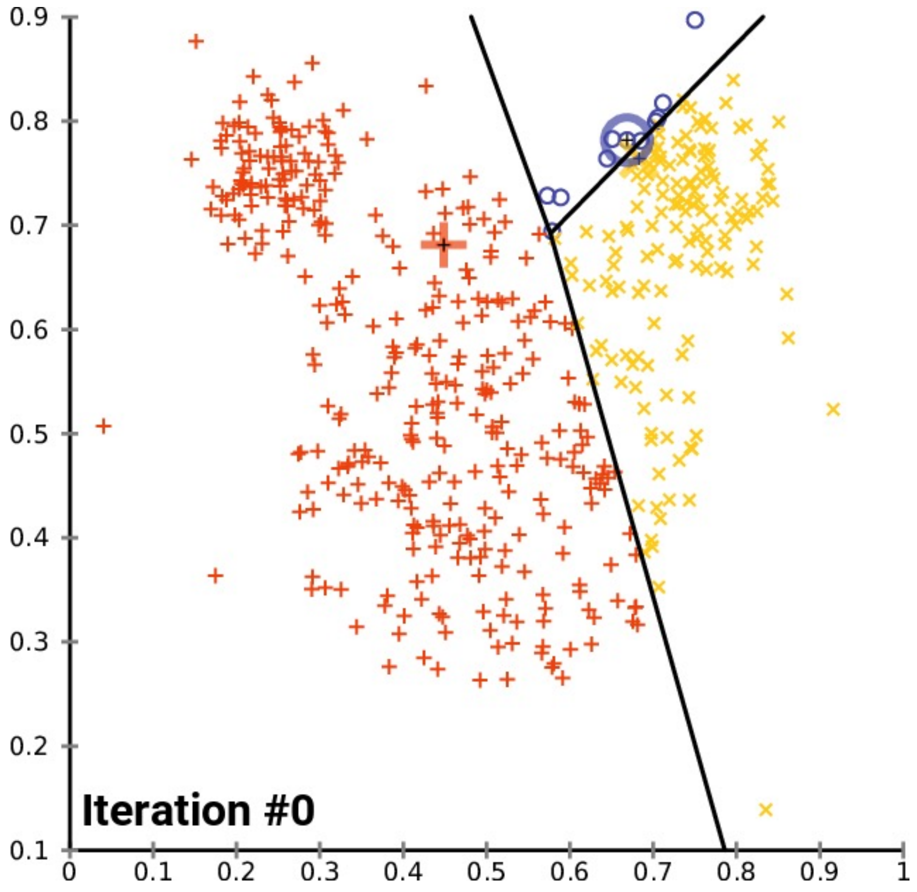
k-means fails



UNIVERSITY OF SAN FRANCISCO

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

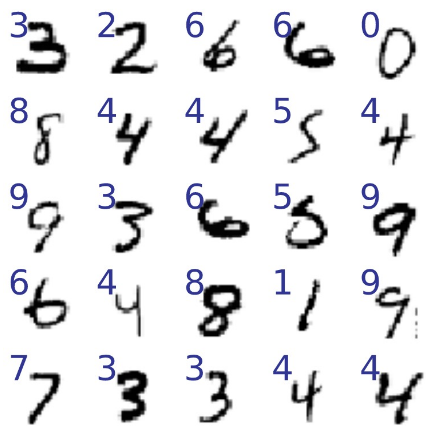
k-means algorithm

Algorithm: *kmeans(X,k)*

```
Select $k$ unique points from $X$ as initial centroids $m_{1..k}^{(t=0)}$ for clusters $C_{1..k}^{(t=0)}$
repeat
    foreach $x \in X$ do
        $j^* = \arg\min_j distance(x, m_j^{(t)})$        (find closest centroid to $x$)
        Add $x$ to cluster $C_{j^*}^{(t+1)}$              (assign $x$ to cluster)
    end
    for $j = 1..k$ do
        $m_j^{(t+1)} = \frac{1}{|C_j^{(t+1)}|} \sum_{x \in C_j^{(t+1)}} x$      (recompute centroids)
    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

```
X = df_digits.drop('digit', axis=1) # get just pixels
y = df_digits['digit']
kmeans = KMeans(k)
kmeans.fit(X)
y_cluster = kmeans.labels
```

- k-means finds k=10 clusters but cluster 5 doesn't usually correspond to the images of fives

Testing MNIST cluster quality

- For all images in each cluster, **cl**, get the true digits from **y**
  - **y_cluster==cl** indicates which images are in cluster **cl**
  - **y[y_cluster==cl]** indicates the true digit of each image in that cluster
  - Then use most common true digit as guess for that cluster's prediction

```
for cl in range(0,k):
    y_true_digits = y[y_cluster==cl].values # convert from class to digit
    most_common_digit = np.bincount(y_true_digits).argmax()
    accur = np.sum(y_true_digits==most_common_digit) / len(y_true_digits)
```

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

Purity and accuracy of clusters for k=10

- Here are true digits for each cluster found by kmeans:

```
[5 8 8 8 9 3 8 8 8 6 5 5 8 8 5 8 5 0 8 8 5 8 0 5 8 8 5 9 8 8] mode = 8, gini = 0.64, accur = 53.0%
[2 2 2 2 2 2 5 2 2 2 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2] mode = 2, gini = 0.18, accur = 90.2%
[3 3 3 3 2 5 2 2 3 3 3 5 3 3 8 0 8 3 5 5 5 8 3 3 3 5 8 3 0 3] mode = 3, gini = 0.64, accur = 53.0%
[4 0 7 9 4 4 4 7 7 9 5 4 4 9 6 4 2 7 9 4 7 4 4 7 7 3 9 4 9 4] mode = 4, gini = 0.72, accur = 35.6%
[0 0 0 0 0 0 5 0 0 0 0 0 0 6 0 0 2 0 5 0 0 5 0 0 0 0 0 0 5 0] mode = 0, gini = 0.38, accur = 77.9%
[7 9 7 9 9 9 4 7 4 9 4 7 4 9 9 9 9 7 7 9 7 4 4 7 9 7 9 4 7 7] mode = 7, gini = 0.69, accur = 43.1%
[6 6 2 6 6 6 6 6 6 6 6 6 6 4 6 6 1 6 6 6 6 2 6 6 6 0 6 6 6 6] mode = 6, gini = 0.26, accur = 85.7%
[0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9] mode = 0, gini = 0.19, accur = 89.6%
[1 1 2 1 1 6 1 5 7 1 7 1 8 4 5 1 5 1 1 2 6 5 1 8 1 5 2 1 1 1] mode = 1, gini = 0.69, accur = 51.6%
[1 1 1 1 1 1 2 1 2 1 1 9 1 9 1 1 1 3 6 3 7 3 1 9 3 1 2 1 1 6] mode = 1, gini = 0.57, accur = 64.5%
Unique labels [0 1 2 3 4 6 7 8], within class avg accuracy 64.4
```

Missing 5, 9!!

- k-means doesn't work that well if we use k=10

Purity and accuracy of clusters for k=20

- Here are true digits for each cluster found by kmeans:

```
[3 5 5 8 3 5 8 3 8 3 3 3 5 3 3 8 3 3 3 5 5 8 5 5 3 5 3 3 8 3] mode = 3, gini = 0.62, accur = 50.3%
[4 9 4 4 5 4 4 9 4 4 4 4 4 4 4 9 4 4 4 9 9 4 9 4 4 4 4 9 4 4] mode = 4, gini = 0.50, accur = 63.4%
[9 9 7 9 9 9 7 7 7 9 7 7 7 7 7 9 7 5 7 4 9 7 9 9 7 4 4 7 7 9] mode = 7, gini = 0.50, accur = 66.3%
[1 1 1 1 1 1 1 2 1 1 1 9 1 1 1 3 6 1 3 1 1 1 6 2 1 1 1 1 1 1] mode = 1, gini = 0.30, accur = 83.6%
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2] mode = 2, gini = 0.09, accur = 95.5%
…
[8 8 8 8 8 8 7 8 8 5 8 8 8 8 8 8 8 2 3 8 8 8 8 8 8 8 8 8 8 8] mode = 8, gini = 0.35, accur = 80.0%
[9 7 9 4 4 9 4 9 9 4 9 7 9 9 9 9 4 9 9 4 7 9 4 9 4 7 9 4 4 4] mode = 9, gini = 0.61, accur = 48.3%
[2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2] mode = 2, gini = 0.13, accur = 93.0%
[7 7 9 4 9 7 9 7 7 9 9 3 7 9 3 9 9 9 7 9 9 7 9 9 4 6 4 9 9 9] mode = 9, gini = 0.71, accur = 38.9%
[7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 2 7 7 7 7 7 7 7 7 7 2 7 7] mode = 7, gini = 0.09, accur = 95.2%
[4 4 6 2 4 7 4 4 4 9 4 4 4 4 4 4 4 7 4 9 4 4 4 4 9 9 7 9 4 4] mode = 4, gini = 0.67, accur = 45.8%
[0 0 0 0 0 0 0 0 0 0 0 2 5 5 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0] mode = 0, gini = 0.20, accur = 89.1%
Unique labels [0 1 2 3 4 5 6 7 8 9], within class avg accuracy 73.6
```

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

Purity and accuracy of clusters for k>20

- k=100:

  Unique labels [0 1 2 3 4 5 6 7 8 9], within class avg accuracy 88.6
- k=200:

  Unique labels [0 1 2 3 4 5 6 7 8 9], within class avg accuracy 90.8
- k=250:

  Unique labels [0 1 2 3 4 5 6 7 8 9], within class avg accuracy 91.4
- Naturally, we'd have to combine these k classes to group into 10 digits

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

k-means application: Breast cancer

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

- 212 cancer, 357 non-cancer
- k-means isn't great; uncertainty is low for one class (0.0056) but high for the other (.4743)
- To the right are the two possible conf matrices, depending on the cluster number chosen by k-means for cancer and for non-cancer

```
kmeans = KMeans(n_clusters=2, init='k-means++')
kmeans.fit(X)
y_pred = kmeans.predict(X)
cancer = np.where(y==0)[0]
benign = np.where(y==1)[0]
print( gini(y_pred[benign]), gini(y_pred[cancer]) )
```

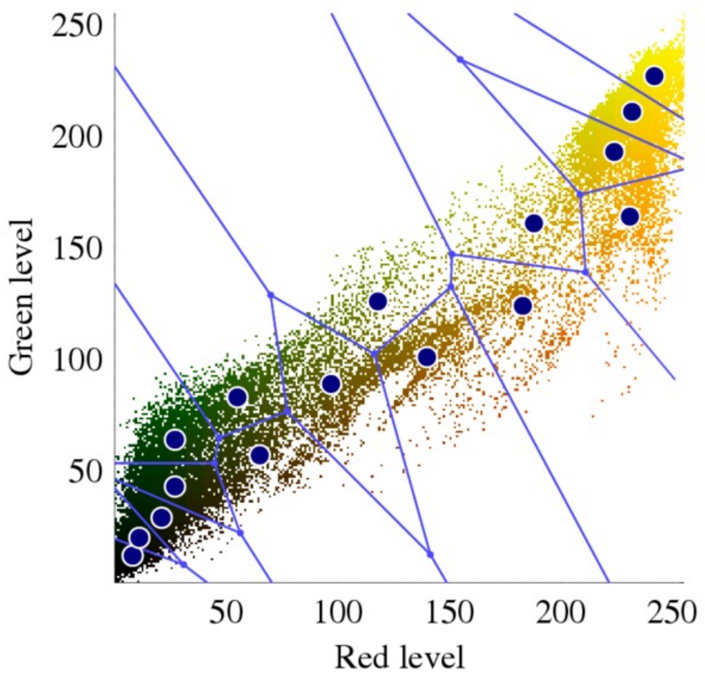
		predicted	
		0	**1**
actual	**0**	356	1
	1	82	130

		predicted	
		0	**1**
actual	**0**	1	356
	1	130	82

k-means application: color quantization

no blue

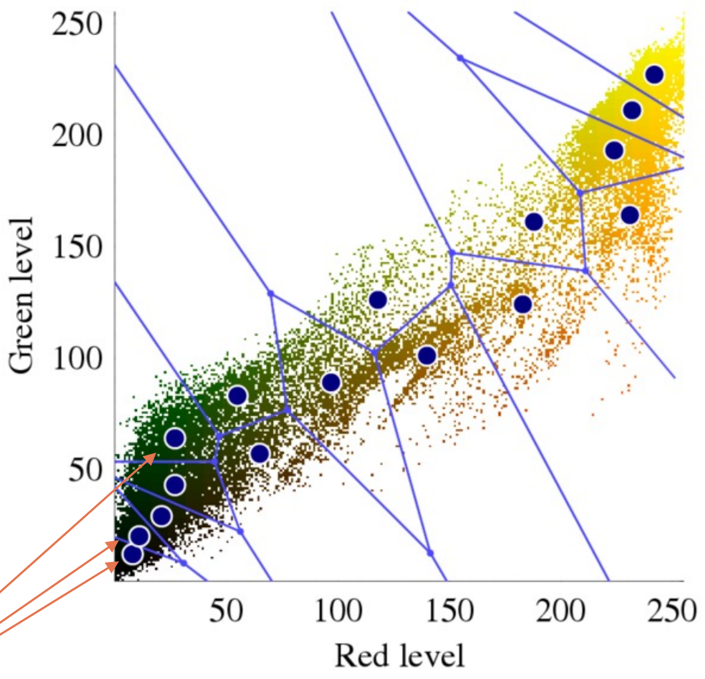
- Color pictures typically use lots of unique colors, possibly 10s of thousands
- Each pixel in the image has Red/Green/Blue colors, 1 byte per RGB = 3 bytes (24 bits)
- Each RGB is a 3D coordinate of color: (R, G, B), with possibly tens of thousands of unique combinations
- Example with just red/green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



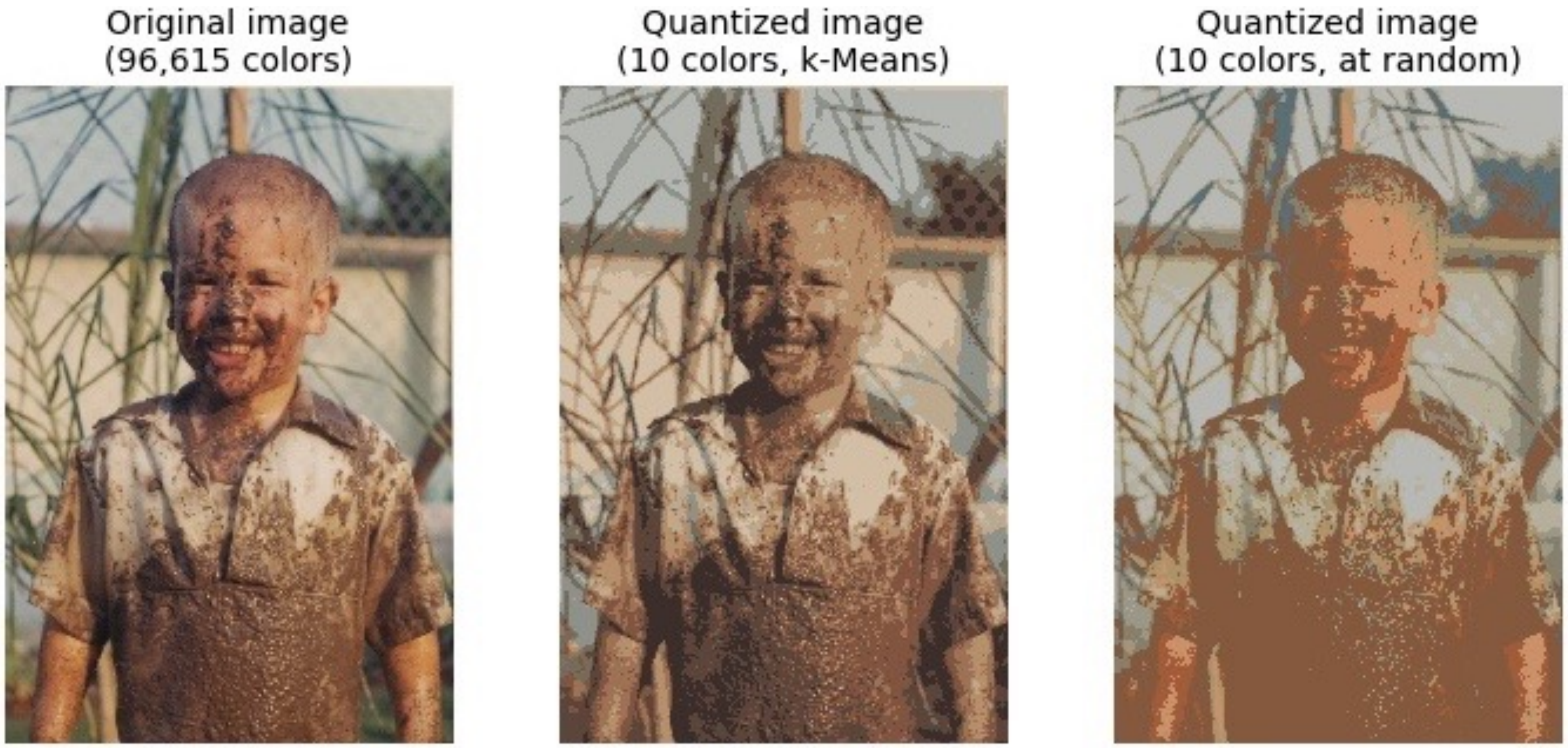
See https://scikit-learn.org/stable/auto_examples/cluster/plot_color_quantization.html
Image from https://en.wikipedia.org/wiki/Color_quantization

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, which is 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

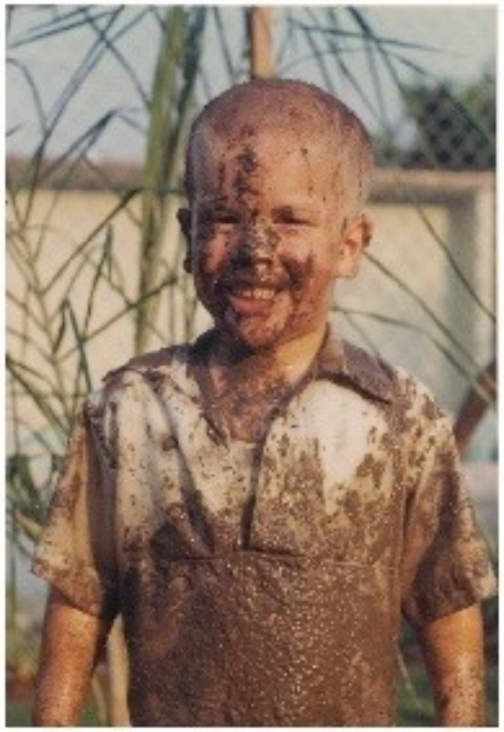


See https://github.com/parrt/msds621/blob/master/notebooks/clustering/kmeans.ipynb

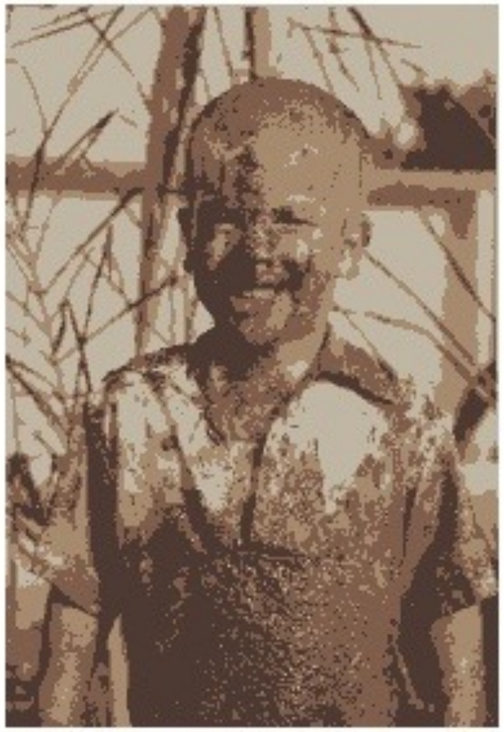
UNIVERSITY OF SAN FRANCISCO

Color quantization example, k=4

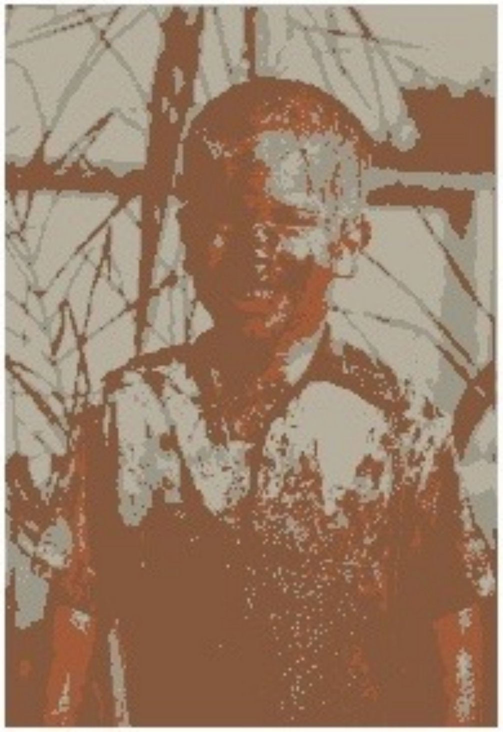
Original image
(96,615 colors)



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 (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

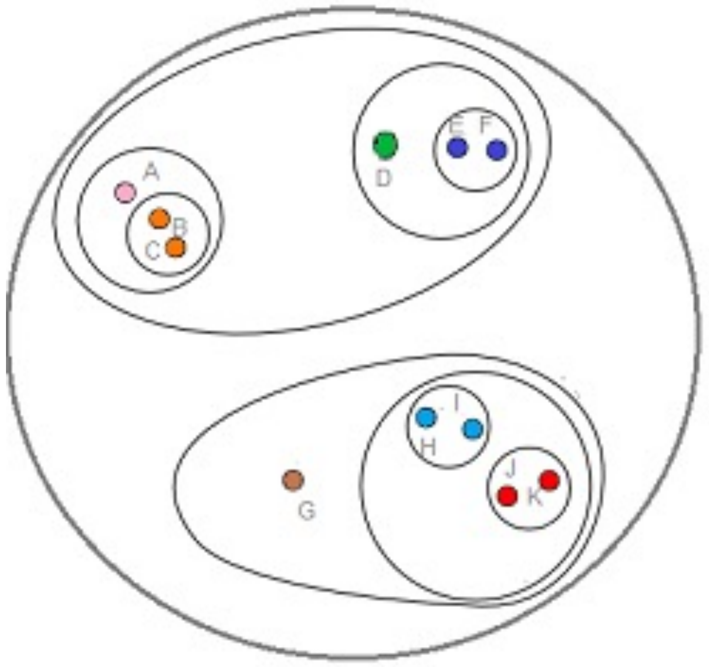
- k-means requires that we specify number of clusters k
- Picking k is usually a problem
- Color quantization and MNIST digits have known k, but few do
- Different starting centroids can lead to very different results
- Each observation is forced into one of the k clusters, but probabilities might be 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 just one cluster left
- Can also stop when distance between clusters are 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

Dendograms

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



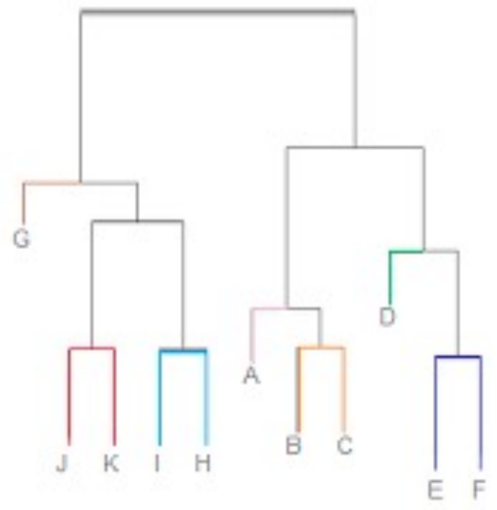


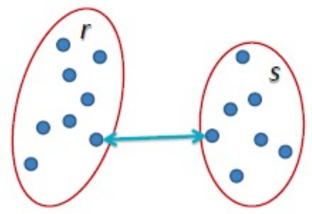
Image from https://www.statisticshowto.datasciencecentral.com/hierarchical-clustering/

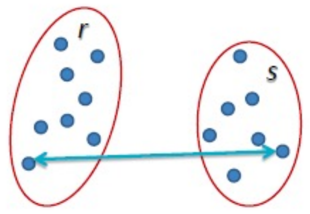
UNIVERSITY OF SAN FRANCISCO

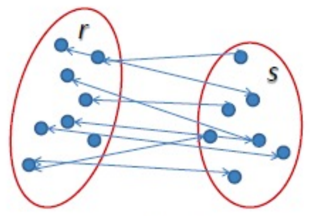
Between-cluster distance metrics (*linkage*)

Warning: statisticians coining terms again!

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-cluster variance; merge pair with smallest prospective variance at each step



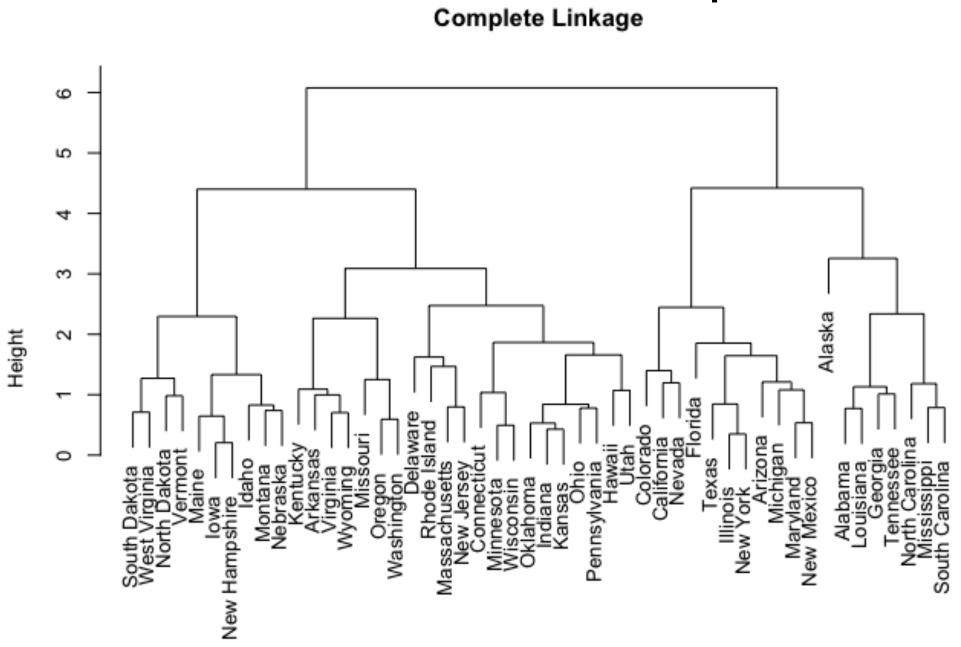




Images from http://saedsayad.com/clustering_hierarchical.htm

Effect of between-cluster-metric

Max distance between points



Min distance between points

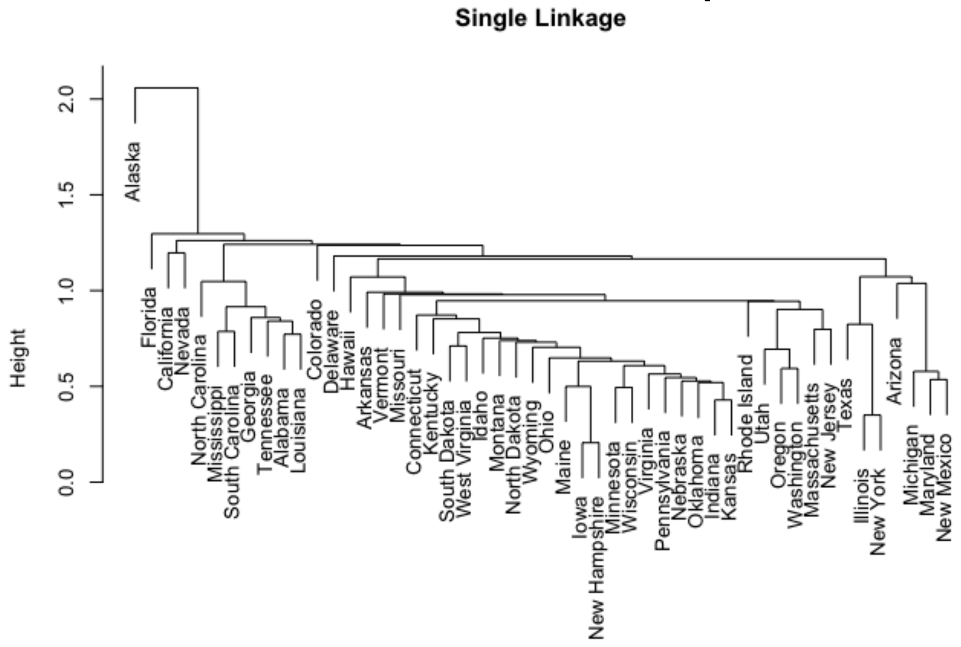


Image from https://uc-r.github.io/hc_clustering

UNIVERSITY OF SAN FRANCISCO

Ward’s method

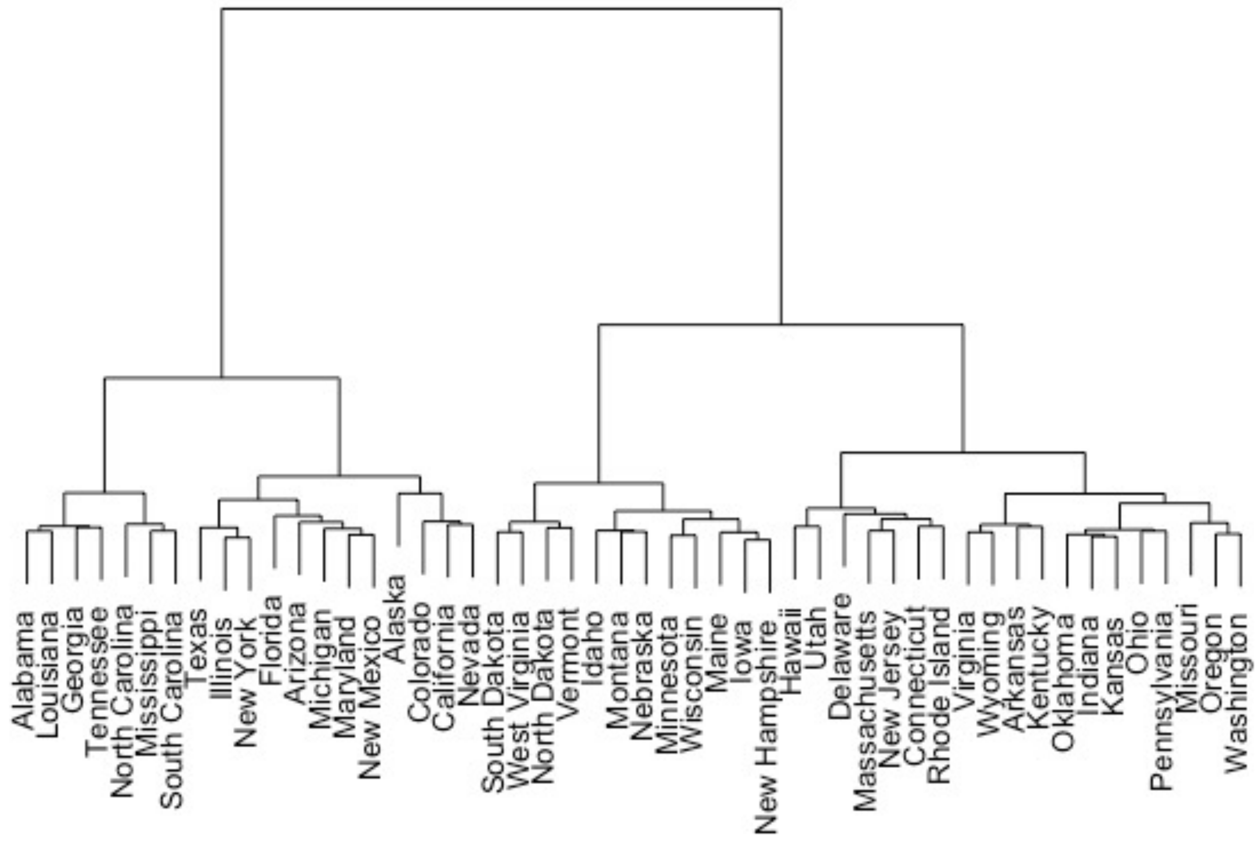


Image from https://uc-r.github.io/hc_clustering

UNIVERSITY OF SAN FRANCISCO

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
- Try your own word embeddings from corpus
- 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

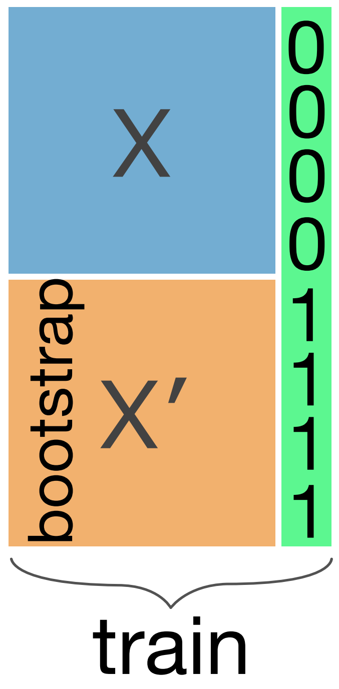
Breiman's RF clustering

- 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 supervised problem
- Then derive 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 <u>univariate</u> distributions…*" of X
   - X' destroys relationships between columns of X
3. Create y to label/distinguish X vs X'
4. Train RF on stacked $[X, X']\rightarrow y$
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 gets 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], len(X), replace=True)
    return X_rand
```

Breiman’s RF conjures up 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)
```

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

Computing RF similarity matrix

```
For each tree in RF
    For each leaf in tree
        Increment similarity for all $x^{(i)}$ and $x^{(j)}$ in leaf (ignoring $X'$ obs.)
Divide each similarity[i,j] by number of leaves to normalize similarities
```

More on RFs for similarity

- **Similarity Forests**
  http://biorxiv.org/cgi/reprint/258699v1
- **Unsupervised Learning With Random Forest Predictors**
  https://horvath.genetics.ucla.edu/html/RFclustering/RFclustering/RandomForestHorvath.pdf