Unsupervised Learning: K-Means

CSCI-P556 Applied Machine Learning
Lecture 22

D.S. Williamson

Agenda and Learning Outcomes

Today's Topics

- **Topics**:
  - Ensemble Learning: Gradient Boosting
  - Unsupervised Learning: K-Means

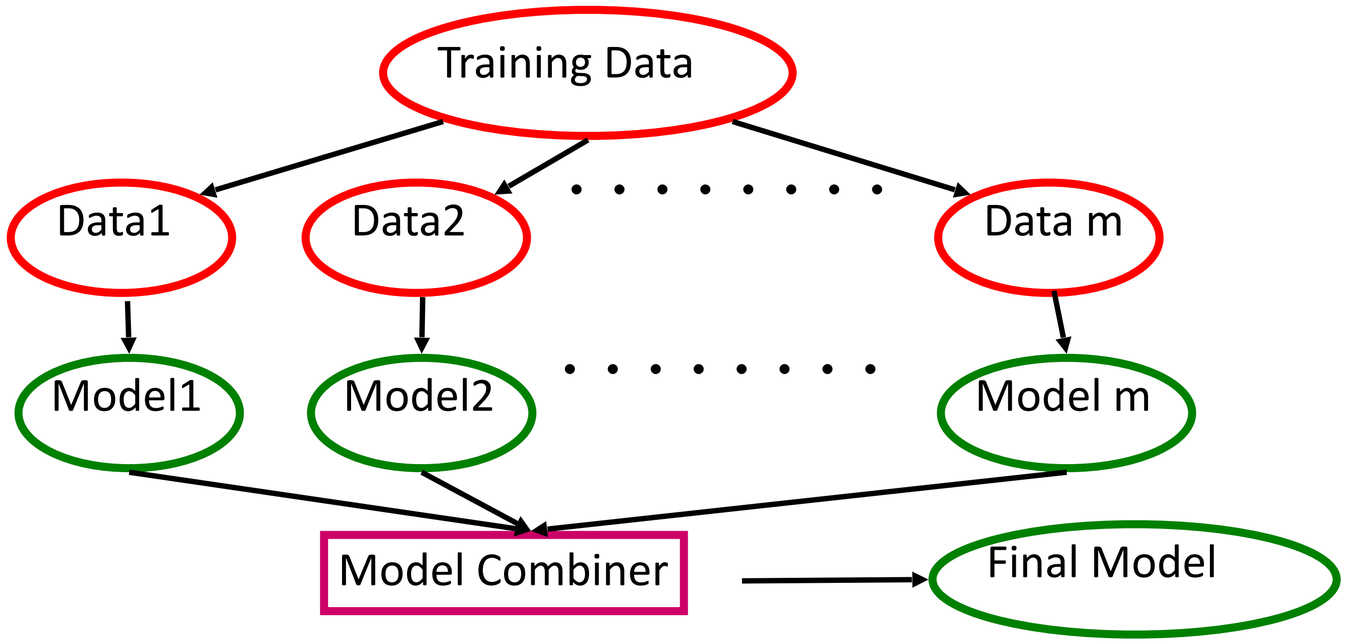
Learning Ensembles

Two approaches

- Perform learning using ***different training data*** or ***different learning algorithms***.
- Combine decisions of multiple definitions, e.g. using weighted voting.

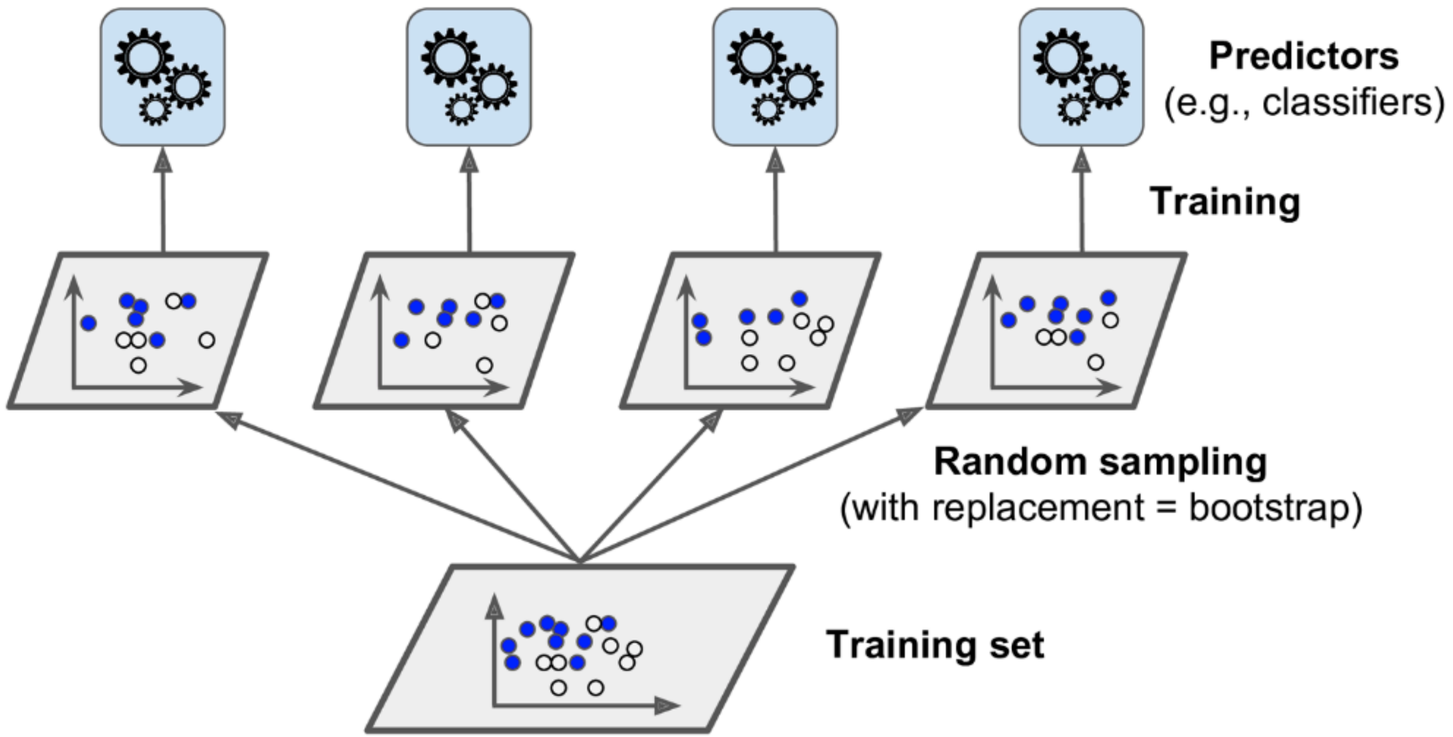
- **When the data varies**, these ensemble learners is either based on **(1) bagging (bootstrap aggregation) or (2) pasting**

- **Key Feature:** They take a single learning algorithm and generate multiple variations (ensembles)



A Depiction of Bagging

- With ***Bagging***, training samples are randomly selected for each variation, but ***sampling is performed with replacement.*** Hence, there may or may not be data overlap across the variations

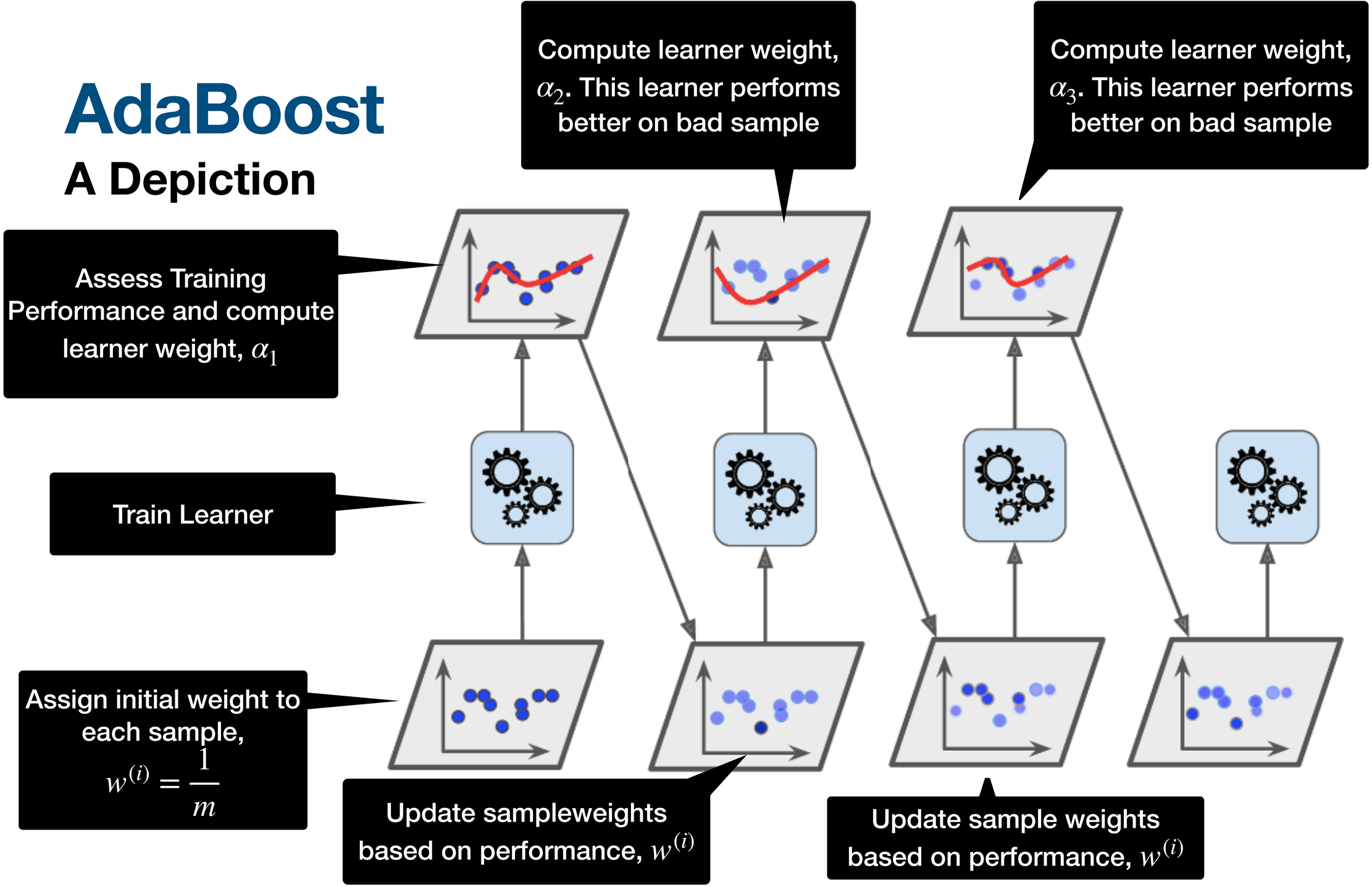


Other Ensemble Learnings

- **Boosting** - sequentially train learning algorithms, where subsequent predictors correct mistakes made by the predecessor. **Two *popular approaches are***
  - *AdaBoost* - Based on sample misclassification/error
  - *Gradient Boost* - Based on learners error
- **Stacking** (stacked generalization) - train a model to perform the aggregation between multiple learners.

AdaBoost

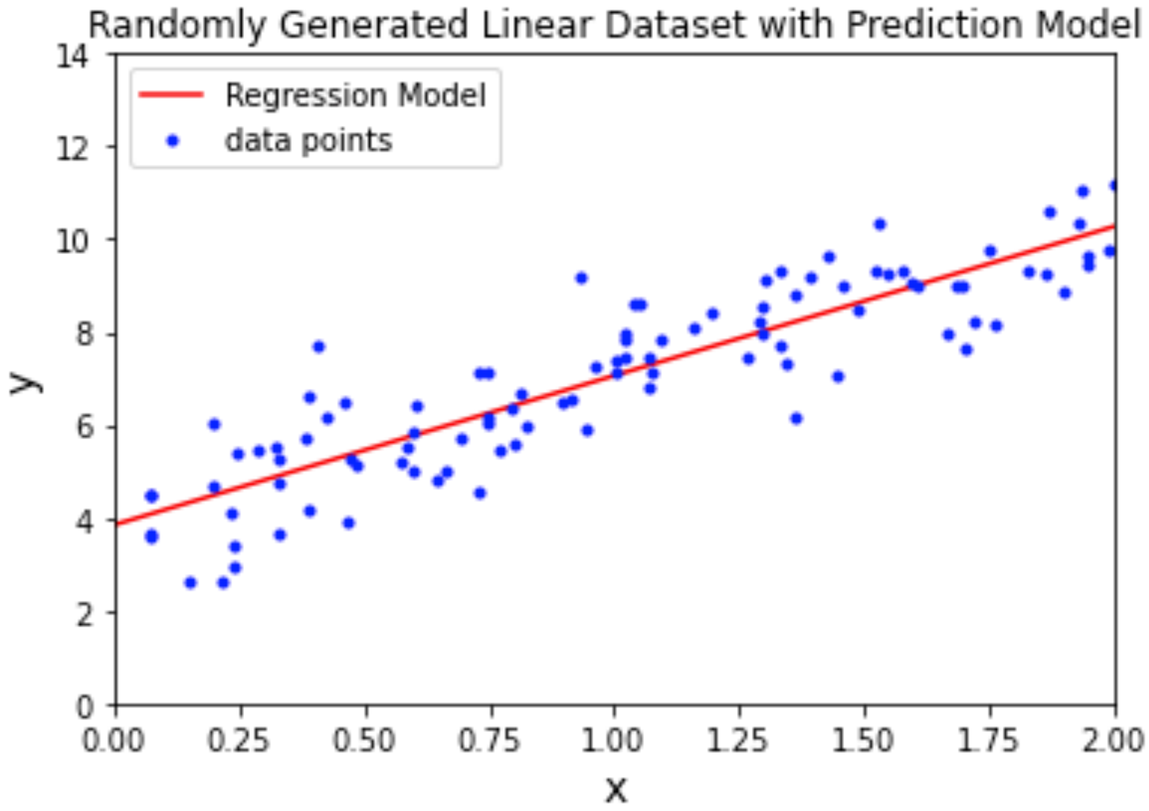
A Depiction



Recall: Linear Regression

AdaBoost

- ***Which points would have the highest weight after training for the first time?***



Gradient Boosting

Fit new learners to residual errors from predecessor

- Train first learner:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

- Compute ***residual error*** then train second learner to predict them

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)
```

- Compute residual error from second learner and train third learner

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)
```

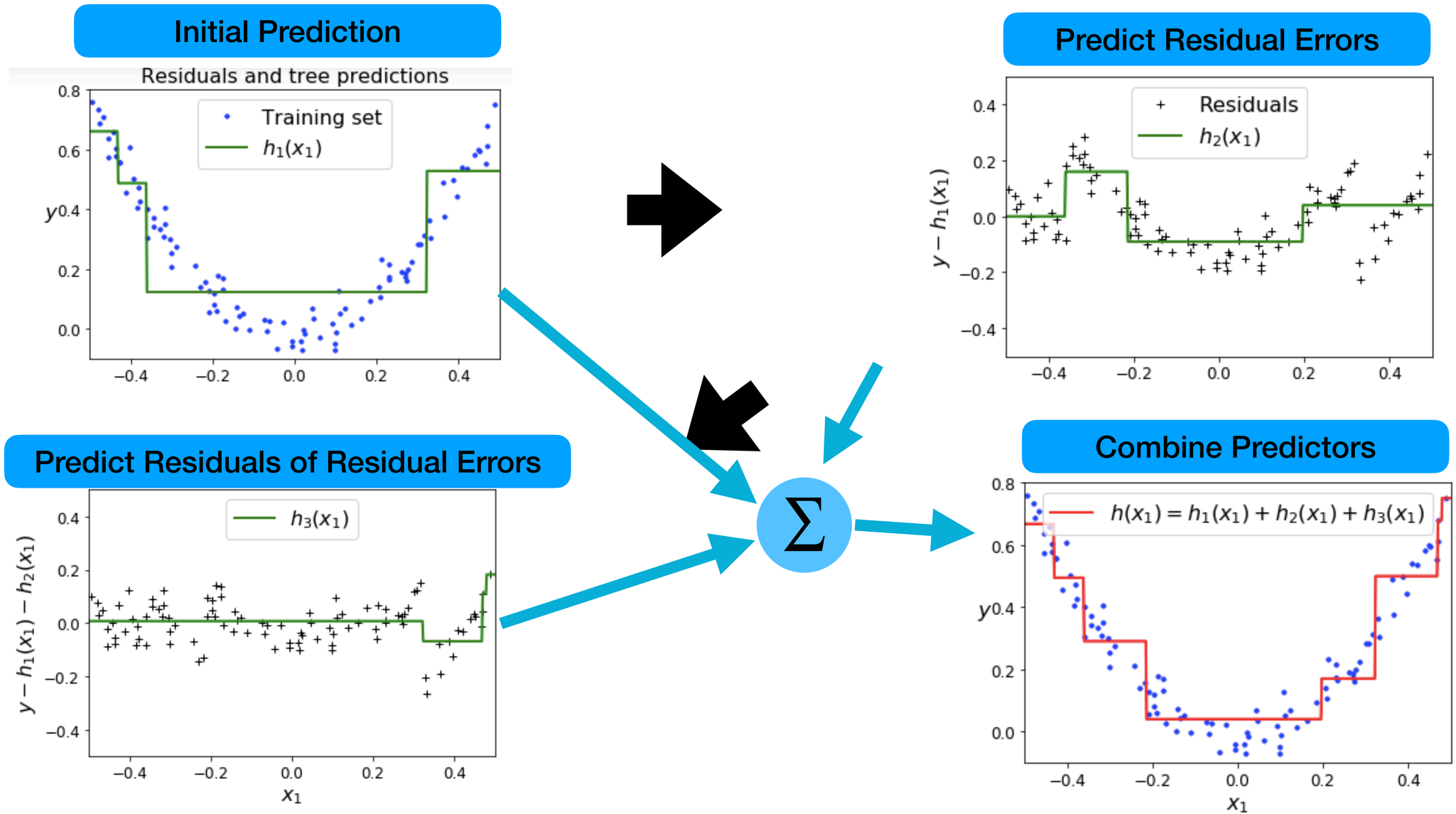
Gradient Boosting

- Make predictions by adding predictions from each learner

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Gradient Boosting

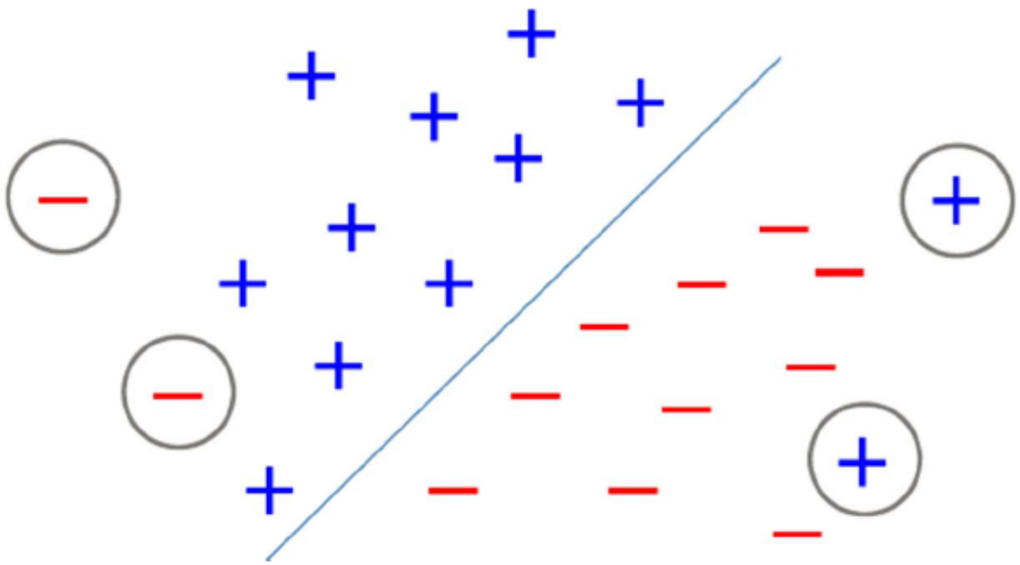
A Visual Example



Pitfall of Boosting

Sensitive to noise and outliers

- The Good: Can identify outliers since focuses on examples that are hard to categorize
- The Bad: Too many outliers can degrade classification performance dramatically increasing time to convergence



Summary: Ensemble Learning

Boosting and Bagging

- **<u>Bagging</u>**
  - Resample data points
  - Weight of each classifier is the same
  - Only variance reduction
  - Robust to noise and outliers

- **<u>Boosting</u>**
  - Re-weight data points (modify data distribution)
  - Weight of classifier vary depending on accuracy
  - Reduces both bias and variance
  - Can hurt performance with noise and outliers

Unsupervised Learning: K-Means

Unsupervised Learning

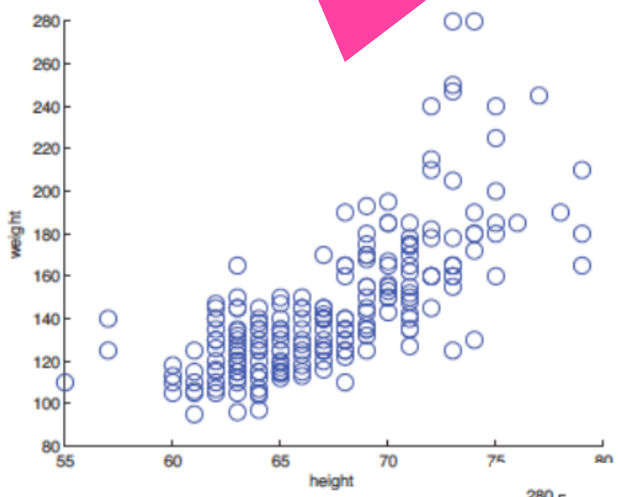
Discover patterns or structure in the data

- Only have (or use) the data information (e.g. ignore labels)

$$D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_N\}$$

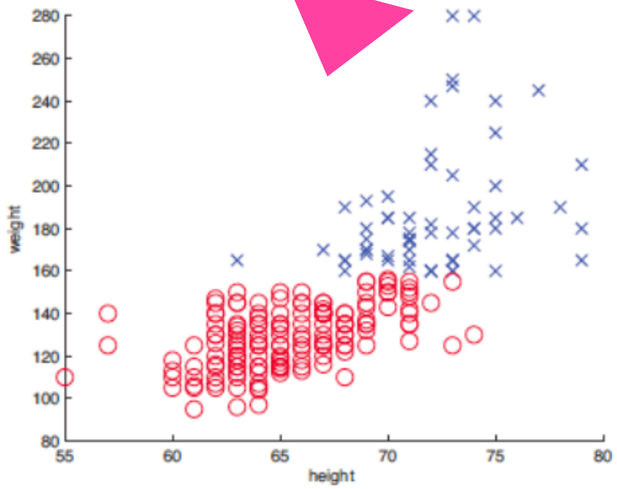
- Examples of unsupervised learning
  - **Clustering**: K-means, vector quantization, Gaussian mixture models
  - **Dimensionality Reduction**: principal components analysis, nonnegative matrix factorization
  - **Topic Modeling:** often used in NLP

Input Data



(a)

Date Clustered into two Groups



(b)

Unsupervised Learning: Clustering

K-Means Algorithms

- **<u>Goal of Clustering</u>**: Find a natural grouping in data so that items in the same cluster are more similar to each other than to items in other clusters

- ***<u>K-Means Clustering</u>***: Divide *N* input patterns into *K* clusters so as to minimize the final variance. In other words, partition patterns into *K* clusters C_j's (j = 1, …,K) to minimize the following cost function

$$J = \sum_{j=1}^{K} \underbrace{\sum_{i \in C_j} ||\mathbf{x}_i - \mathbf{u_j}||}_{\text{Variance of data in j-th cluster}}$$

Sum over all clusters

where $\mathbf{u}_j = \frac{1}{||C_j||} \sum_{i \in C_i} \mathbf{x}_i$ is the mean (center) of cluster j

- Cluster is represented by ***<u>centroid</u>*** (e.g., $\mathbf{u_j}$, mean of data in cluster)

K-Means Algorithm

Iterative Steps

1. Choose a set of ***K* cluster centers randomly** from the **input data; or randomly initialize**

2. Assign the *N* input patterns (individually) to the *K* clusters using the squared Euclidean distance rule. **x** is assigned to C_j if:

$$||\mathbf{x}-\mathbf{u}_j||^2 \leq ||\mathbf{x}-\mathbf{u}_i||^2 \text{ for } i \neq j$$

3. Update cluster centers based on data that is assigned to each cluster

The number of data samples in the j-th cluster

$$\mathbf{u}_j = \frac{1}{||C_j||} \sum_{i \in C_j} \mathbf{x}_i$$

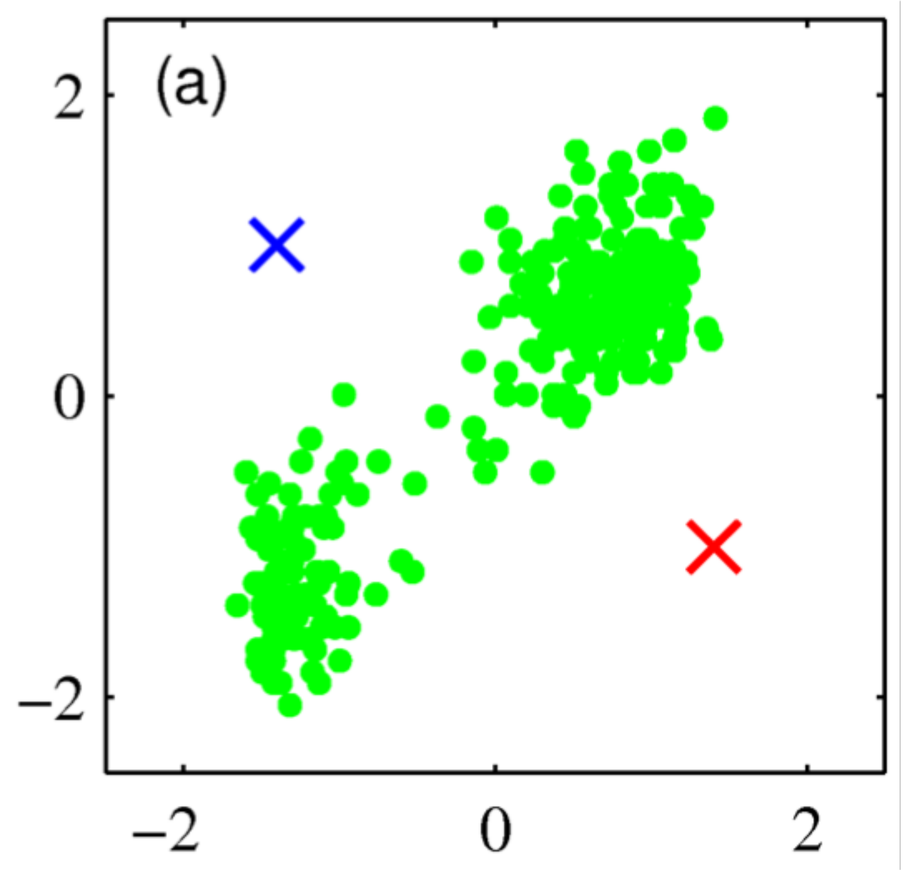
4. If any cluster center changes, go to step 2; otherwise stop. Can also specify a tolerance threshold for stopping

The K-means algorithm always converges, but the global minimum is not assured

K-Means Illustration

(Randomly) Initialize Cluster Centroids (or Centers)

- Clustering data into two clusters using K-means
  - Blue X is initial center of one cluster
  - Red X is initial center of the other



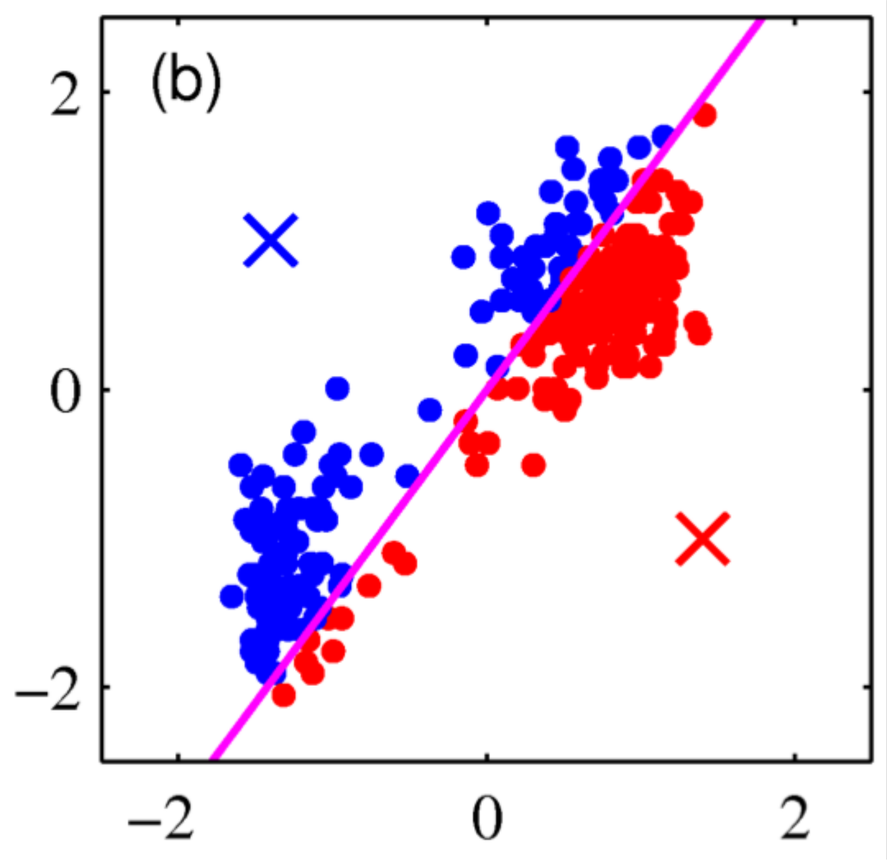
From Bishop (2006)

K-Means Illustration

Assign Data to one of the Clusters

- Clustering data into two clusters using K-means
  - Blue X is center of one cluster
  - Red X is center of the other

- After the first iteration, some of the data is put in the Red cluster, while the other is put in the Blue cluster
  - This is based on closeness

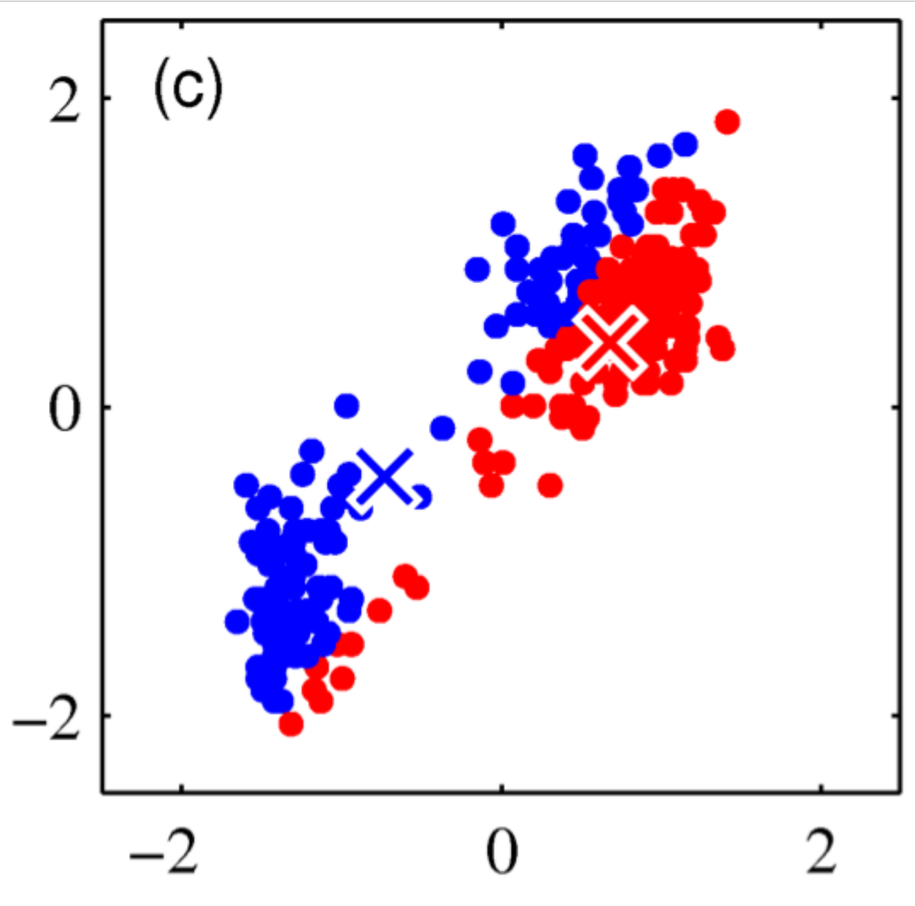


From Bishop (2006)

K-Means Illustration

Update Cluster Centroids based on recent Assignment

- Clustering data into two clusters using K-means
  - Blue X is center of one cluster
  - Red X is center of the other
- After the first iteration, some of the data is put in the Red cluster, while the other is put in the Blue cluster
  - This is based on closeness
- Update Cluster Centroids

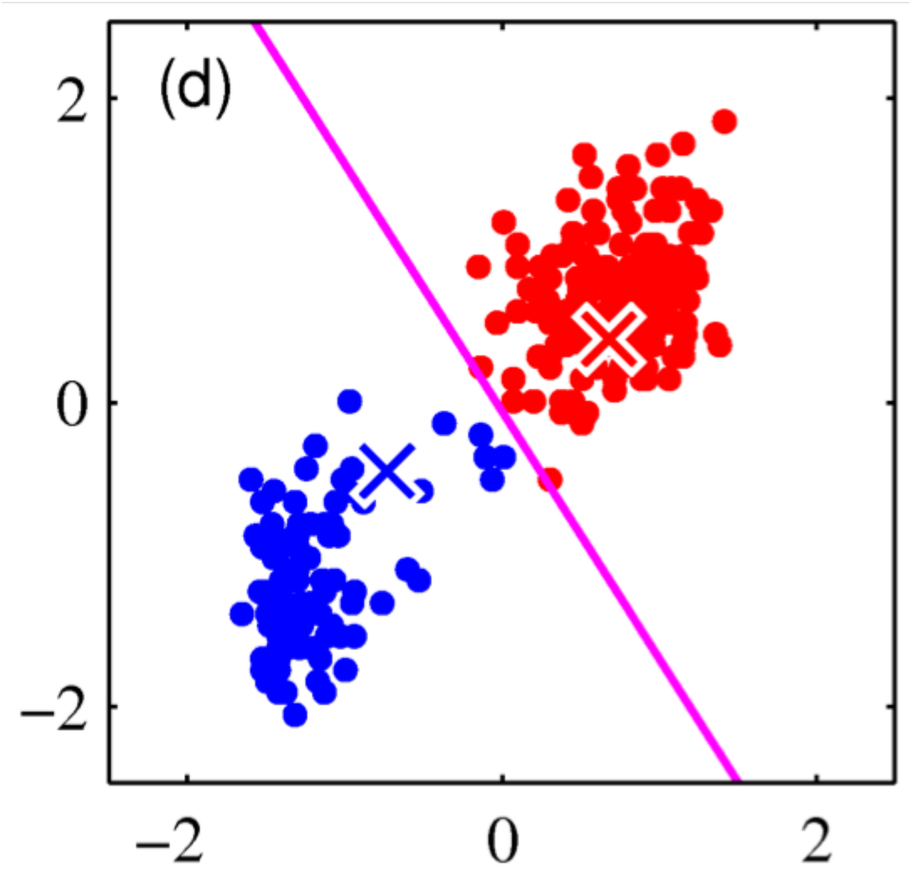


From Bishop (2006)

K-Means Illustration

Re-assigning Data Samples to Clusters

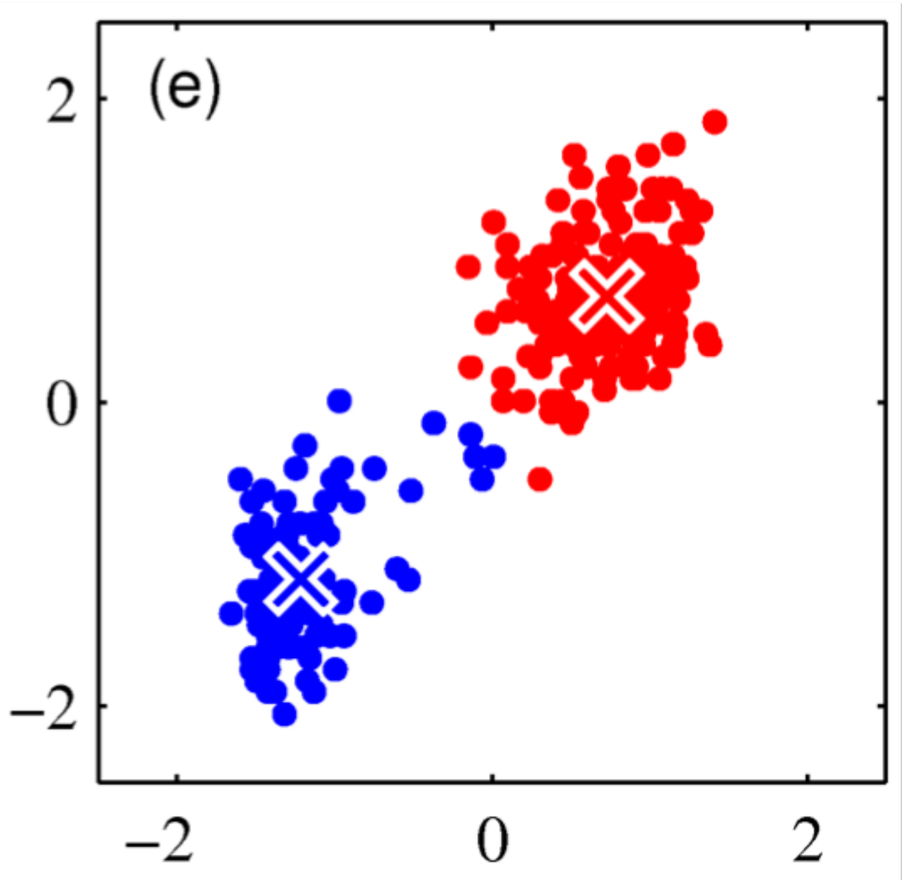
- The cluster centroids changed, so we need to re-assign all of the points to the closest cluster



From Bishop (2006)

K-Means Illustration

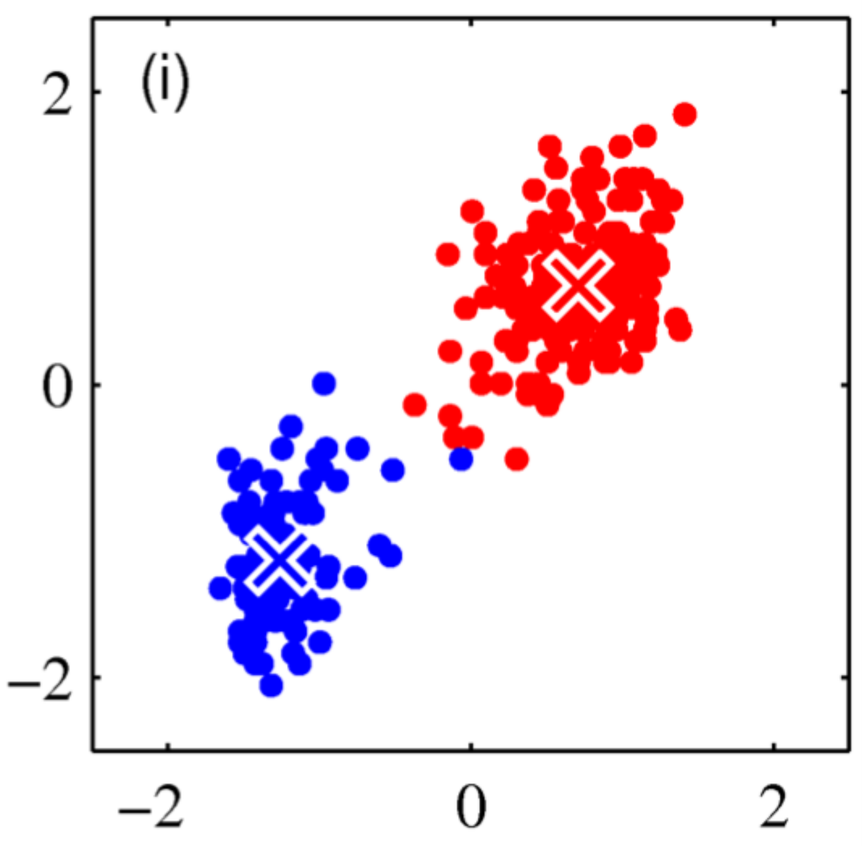
- Update Cluster Centroids again



From Bishop (2006)

K-Means Illustration

- Cluster centroids do NOT change again, so we can stop the algorithm



From Bishop (2006)

K-Means in Python

sklearn.cluster.KMeans

class `sklearn.cluster.`**`KMeans`**(*n_clusters=8, *, init='k-means++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='deprecated', verbose=0, random_state=None, copy_x=True, n_jobs='deprecated', algorithm='auto'*)

[source]

Parameters:

n_clusters : ***int, default=8***

The number of clusters to form as well as the number of centroids to generate.

init : ***{'k-means++', 'random'}, callable or array-like of shape (n_clusters, n_features), default='k-means++'***

Method for initialization:

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose `n_clusters` observations (rows) at random from data for the initial centroids.

If an array is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

If a callable is passed, it should take arguments X, n_clusters and a random state and return an initialization.

n_init : ***int, default=10***

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter : ***int, default=300***

Maximum number of iterations of the k-means algorithm for a single run.

tol : ***float, default=1e-4***

Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
       [ 1.,  2.]])
```

Summary of K-Means

- Advantages:
  - Easy to implement
  - Computationally efficient compared to other clustering algorithms

- Disadvantages
  - Must specify number of clusters, K, *a priori*. Bad choice results in poor clustering
  - Clusters can be empty
  - May not reach global minimum (depends on initialization)
  - Features MUST be scaled (e.g. normalization or min-max scaling)

Overcoming Issues with K-Means

- **Bad initialization or Slow Convergence:**
    1. Run K-means algorithm multiple times and choose the best performing model
    2. Place initial centroids far away from each other via the ***K-means++ algorithm***
        1. Randomly pick first centroid from data samples
        2. For each remaining data sample, compute its distance to centroid
        3. Select the next centroid from the data samples, so the probability of choosing a point as centroid is proportional to the distance (e.g. pick data sample that is farthest from the nearest centroid
        4. Repeat these steps until K centroids have been determined
        5. Perform K-Means to cluster points based on above initial K centroids

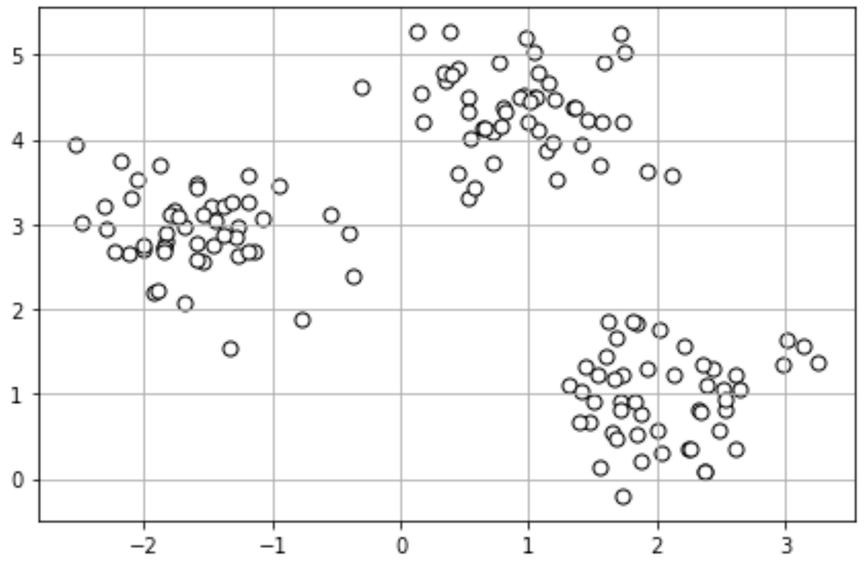
Overcoming Issues with K-Means

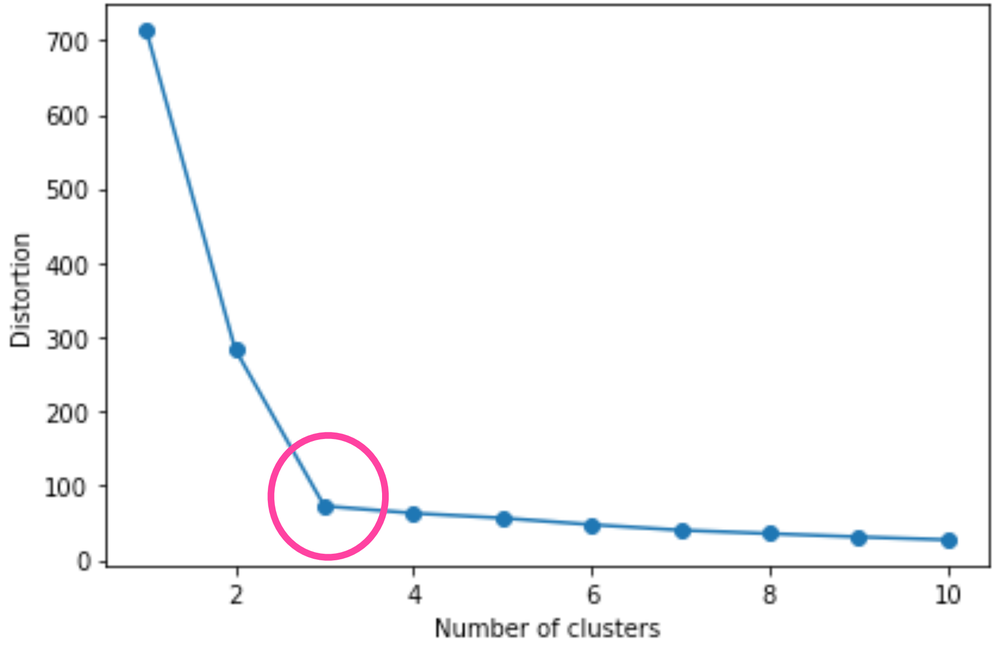
Determining the number of clusters

- Use the ***elbow method*** to determine the appropriate value for K. **Idea**:
  - Run K-means for different values of K
  - For each value of K, compute the sum of the squared error (e.g. total variance across all clusters)
  - Plot this error as a function of K
  - Identify the ***elbow*** in the plot, since beyond this point the distortions begin to increase (e.g. not desired).
  - The elbow becomes the value of K for this dataset.

Overcoming Issues with K-Means

Determining the number of clusters





- Try different values for K and compute error

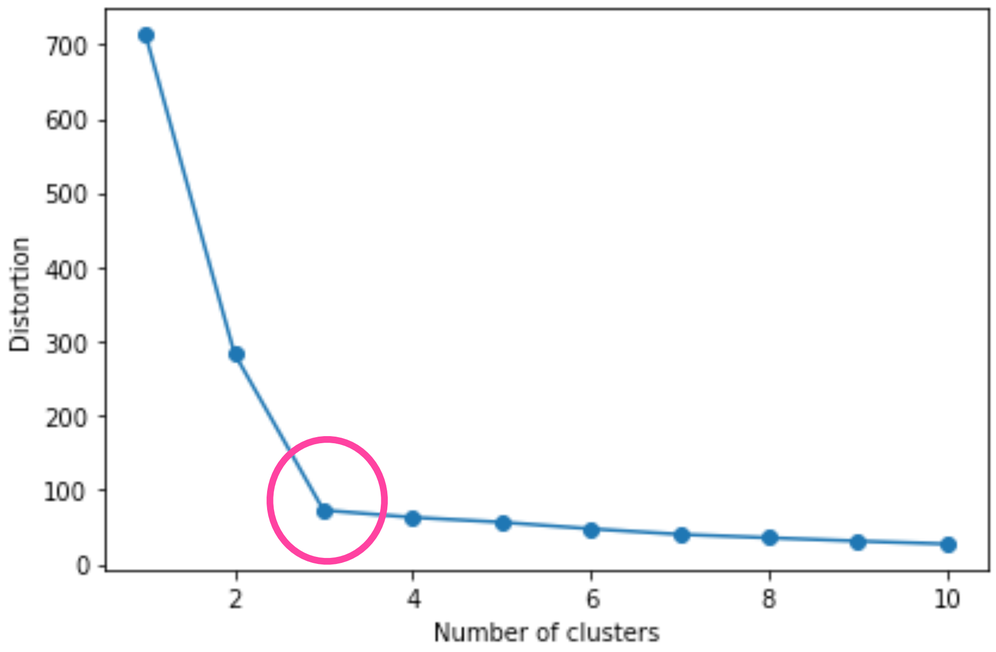
```
distortions = []
for i in range(1,11):
    km = KMeans(n_clusters=i, init='k-means++', n_init=10, max_iter=300, random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)

plt.plot(range(1,11),distortions,marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.tight_layout()
plt.show()
```

- Identify the elbow at K=3
- K=3 is the good choice

Occam's Razor

- **The best scientific model is the simplest that is consistent with the data**
  - In our case, it translates to the principle that a learning machine should be large enough to approximate the data well, but not larger

- Occam's razor is a general principle governing machine learning



Next Class

More on Unsupervised Learning: Gaussian Mixture Models (GMMs)