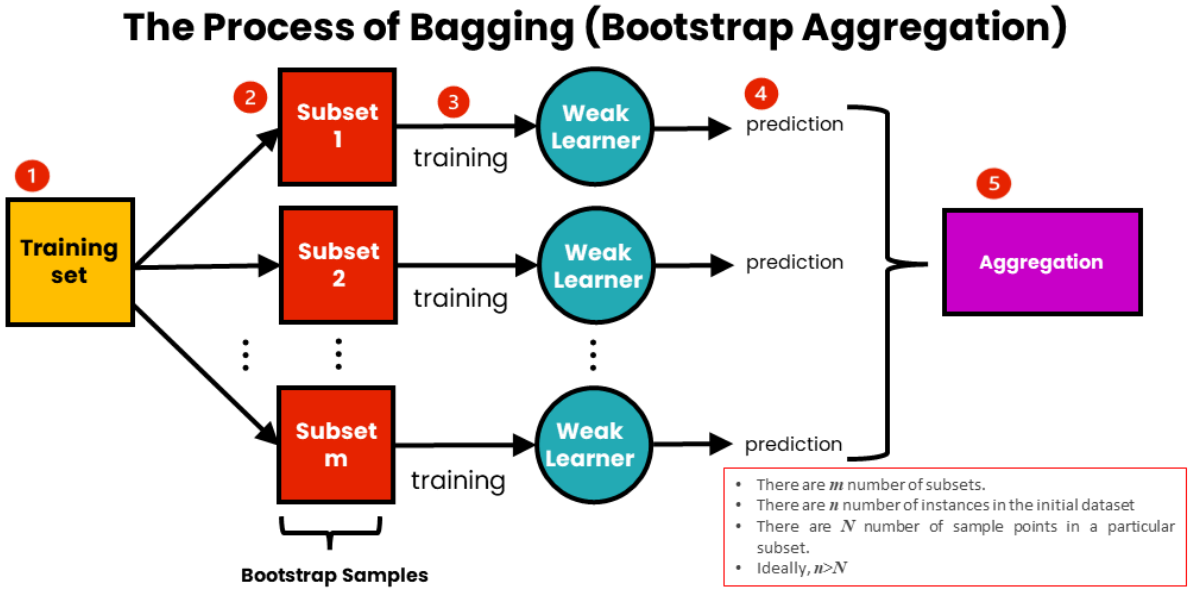
Ensemble Learning

In machine learning, ensemble learning is a powerful technique that combines predictions from multiple models (often called "weak learners") to create a single, more robust and accurate model (the "ensemble"). It's like having a team of experts working together to make a better decision.

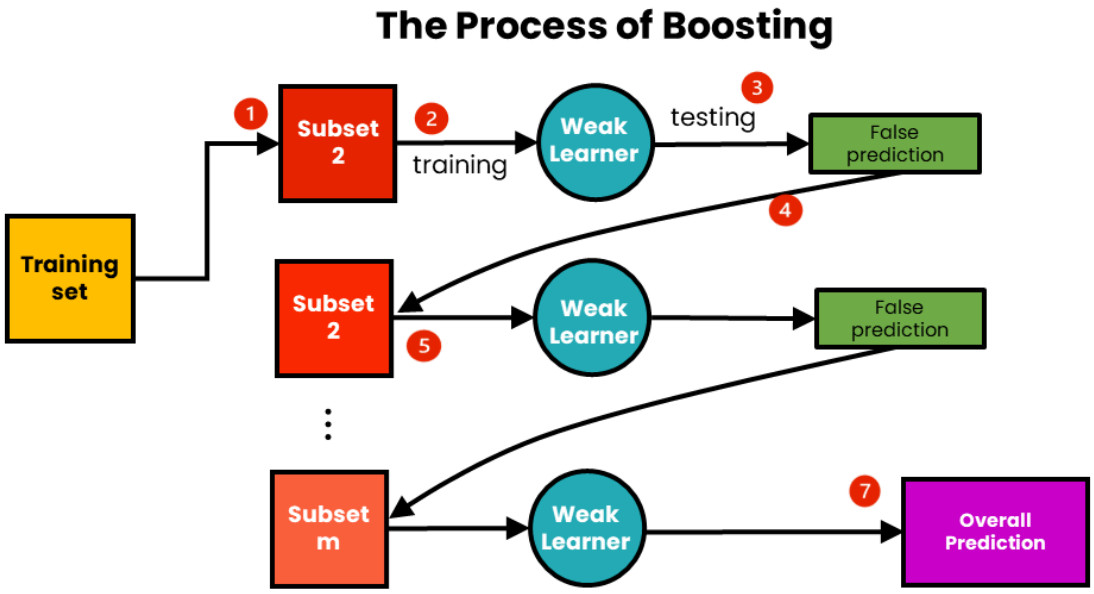
There are two main approaches to ensemble learning:

- **Bagging (Bootstrap aggregating):** Here, multiple models are trained on different subsets (with replacement) of the original data. This injects diversity into the ensemble, making it less prone to overfitting on the training data. Examples of bagging methods include Random Forests and Bagging Classifiers.



- **Boosting:** In boosting, models are trained sequentially. Each model focuses on the errors made by the previous model, improving the overall accuracy

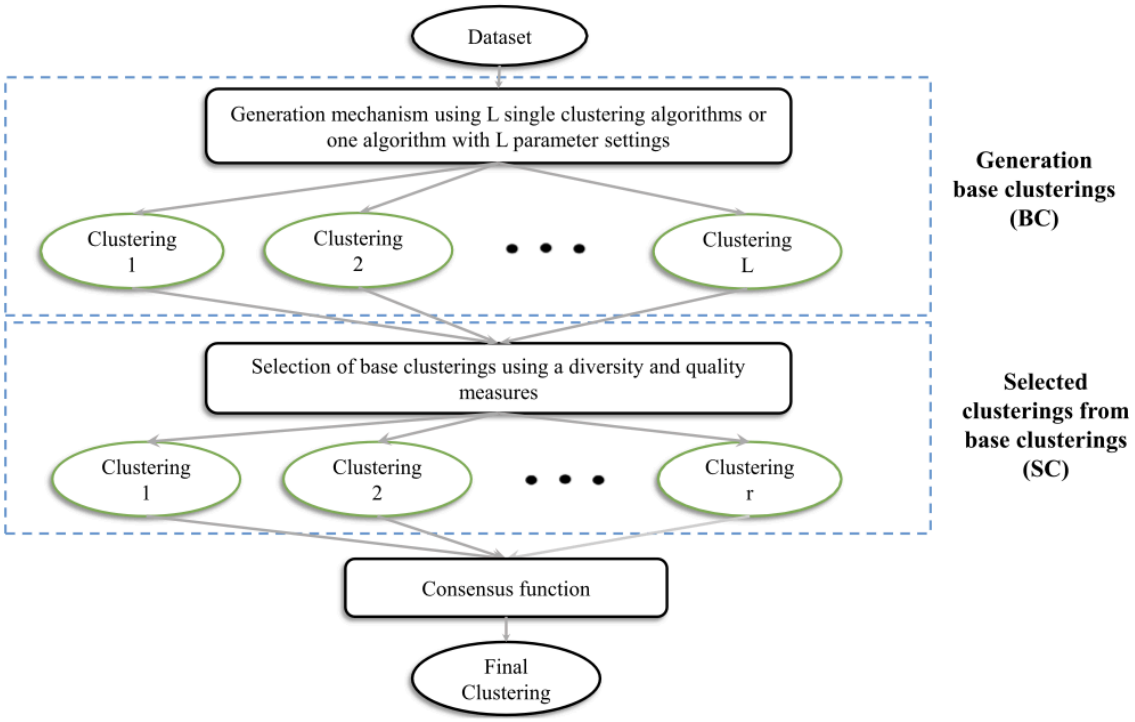
step-by-step. Gradient Boosting and AdaBoost are popular boosting algorithms.



Ensemble Clustering

Ensemble clustering, also known as consensus clustering, applies the ensemble approach to the unsupervised learning task of clustering. Here's how it works:

Ensemble clustering, like other ensemble methods, can be broken down into two main phases:



Phase 1: Generating Diverse Base Clusterings

1. **Data Preprocessing:** As in most machine learning tasks, you might need to clean, normalize, or transform your data for better clustering results.

2. **Run Multiple Clustering Algorithms:** The core of this phase is to generate a collection of different clusterings for the same data. This diversity is crucial for a robust ensemble. Here's how you can achieve it:

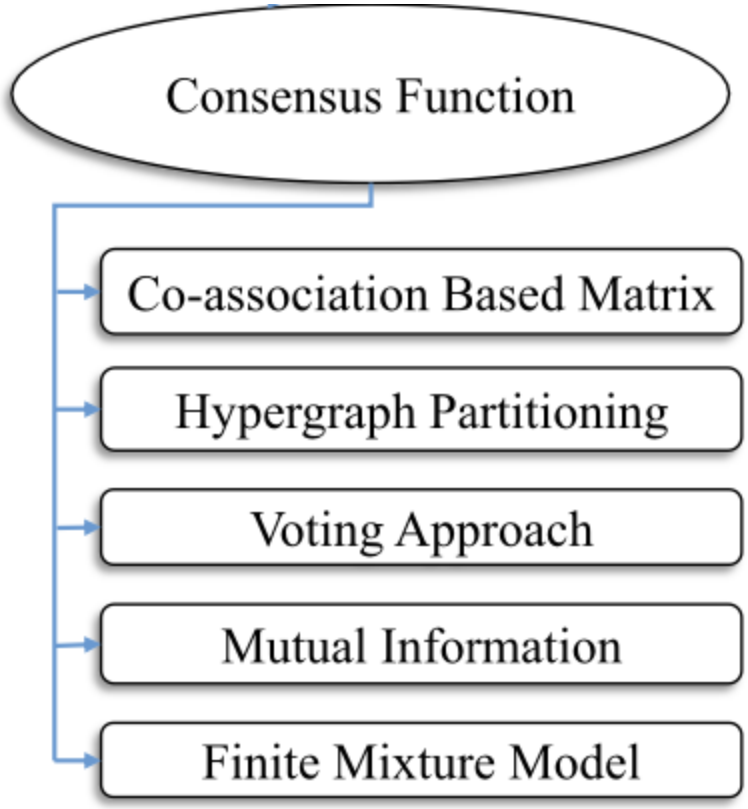
    - **Use Different Clustering Algorithms:** Apply a variety of clustering algorithms like K-means, hierarchical clustering, DBSCAN, etc., each with potentially different parameter settings.

    - **Run the Same Algorithm with Variations:** Even with the same algorithm (e.g., K-means), you can introduce diversity by using different random initializations or varying the number of clusters (k) in each run.
    - **Resample the Data:** Apply clustering algorithms to different subsets of the data created by bootstrapping (sampling with replacement) or subsampling (sampling without replacement).
3. **Store the Results:** You'll end up with a collection of base clusterings, where each base clustering represents a way of grouping the data points. Each base clustering might have a different number of clusters and different data point assignments.

Phase 2: Consensus Function and Final Clustering

1. **Consensus Function:** This is where the magic happens! The consensus function takes all the base clusterings as input and aims to find a single, "consensus" clustering that best reflects the agreement across the diverse base clusterings.
2. **Final Clustering:** Based on the chosen consensus function, a final clustering is generated. This clustering represents a more robust and potentially more accurate grouping of the data points compared to any single base clustering.

five main categories of consensus functions used in clustering ensembles:



1. **Coassociation-Based Matrix Approach:**
    - This approach focuses on pairwise similarities between data points. It calculates a coassociation matrix that measures how often data points are grouped together across the base clusterings.
    - The final clustering aims to group points with high coassociation values, indicating consistent co-occurrence in the base clusterings.
    - Examples include Average Linkage and Ward's method, which are commonly used for hierarchical clustering.
2. **Hypergraph Partitioning Approach:**
    - This method represents data points as vertices and co-occurrences in clusters across the base clusterings as hyperedges (edges that connect more than two vertices).

- The goal is to partition the hypergraph into subgraphs such that vertices within a subgraph are frequently co-clustered across the base clusterings.
- Algorithms like Hypergraph Spectral Clustering (HSPC) and Affinity Propagation (AP) fall into this category.

3. **Voting-Based Approach:**
    - This is a simple and intuitive approach. It treats each base clustering as a "vote" for how data points should be grouped.
    - The final clustering is then determined by the most frequent assignment for each data point across the base clusterings (majority vote).
    - This approach can be effective when the base clusterings have a high degree of agreement.
4. **Mutual Information-Based Approach:**
    - This method leverages the concept of mutual information, which measures the dependence between two variables (data points in this case).
    - It calculates the mutual information between data points based on their co-clustering across the base clusterings.
    - The final clustering aims to group points with high mutual information, indicating a strong relationship between them in terms of co-occurrence in the base clusterings.
5. **Finite Mixture Model (FMM):**
    - This approach assumes the data arises from a mixture of underlying distributions (components), each representing a cluster.
    - The FMM estimates the parameters of a mixture model that best fits the data, considering the information from all the base clusterings.
    - The final clustering is then derived from the estimated component memberships in the FMM.

Benefits of Ensemble Clustering:

- **Improved Accuracy:** By aggregating results from diverse algorithms, ensemble clustering can often achieve better cluster quality compared to a single algorithm.

- **Reduced Sensitivity:** Different algorithms have different strengths and weaknesses. Ensemble methods can mitigate the impact of these weaknesses, leading to more robust clusters.

Some Popular Consensus Algorithm base on Each approach:

Co-association matrix:

- **Combining Multiple Clusterings Using EAC**
- **CE Based on Normalized Edges**
- **Pairwise Similarity Matrix for CE**
- **Probability Accumulation Algorithm**
- **Ensemble Clustering by Matrix Completion (ECMC)**
- **Density-based similarity matrix construction**:
- **Weighted co-association matrices**

Hypergraph partitioning:

Cluster-based Similarity Partitioning Algorithm (CSPA)

The clustering ensemble model based on such consensus function can also be described by a pseudo code:

Input:

- *a set of input partitions* $\{P_1, P_2, \ldots, P_T\}$, *with number of partitions* T
- *The graphic-based clustering algorithm METIS*

for $t = 1$ *to* T
    Find the number of clusters k_t *in partition* P_t
    for $i = 1$ *to* k_t
        construct the hyper-edge h_i
    end for
end for
Compute the adjacency matrix $H = \{h_i\} \sum_{i=1} k_t$
Compute the co-association matrix $\mathbf{S} = HH^T$
$K = \max(k_t)$
$P_{consensus} = METIS(S, K)$
Output: the final clustering $P_{consensus}$.

Input:

- A set of input partitions: {P1, P2, ..., PT}, where T is the number of partitions. These partitions represent the results of clustering the data from different algorithms or runs.

Algorithm:

1. **Iterate through each partition (Pt):**
    - For each partition Pt (from i = 1 to T):
        - Find the number of clusters (kt) within that partition Pt.
2. **Construct hyperedges:**
    - For each data point within a partition Pt (from i = 1 to kt):
        - Construct a hyperedge (hi) that connects the data point to all other data points belonging to the same cluster within Pt.
3. **Compute Adjacency Matrix (H):**
    - Build the adjacency matrix (H). This matrix represents the connections between data points based on the constructed hyperedges.
        - An element (h_ij) in the matrix indicates the number of partitions where data points i and j belong to the same cluster.
4. **Compute Co-association Matrix (S):**
    - Calculate the co-association matrix (S). This matrix captures the overall similarity between data points across all partitions.
        - S is computed by multiplying the transpose of the adjacency matrix (H^T) with the adjacency matrix (H).

        - Apply a normalization factor (K), which is the maximum number of clusters across all partitions (max(kt)).

5. **Consensus Clustering with METIS:**

    - Utilize the METIS algorithm (a graph-based clustering algorithm) on the co-association matrix (S) and the normalization factor (K).
    - METIS will identify the final consensus clustering (P_consensus) based on the similarities captured in the co-association matrix.

Output:

- The final consensus clustering (P_consensus) that represents the most consistent grouping of data points across all input partitions

A pseudo code is also given to describe this consensus function:
Input:

- *a set of input partitions* $\{P_1, P_2, \ldots, P_T\}$*, with number of partitions* T
- *The graphic-based clustering algorithm HMETIS*

for $t = 1$ *to* T
    Find the number of clusters k_t *in partition* P_t
    for $i = 1$ *to* k_t
        construct the hyper-edge h_i
    end for
end for
Compute the adjacency matrix

$$H = \{h_i\} \sum_{i=1} k_t$$
$$K = \max(k_t)$$
$$P_{consensus} = HMETIS(H, K)$$
Output: the final clustering $P_{consensus}$.

Inputs:

- `{P1, P2, ..., PT}`: A set of `T` input partitions, where each `Pi` represents a cluster solution from a different base clustering algorithm.
- `HMETIS`: A reference to the hypergraph partitioning algorithm (replace with your preferred implementation if not using HMETIS).

Steps:

1. **Iterate Over Base Clusterings:**
    - `for t = 1 to T`: The loop iterates through each base clustering (`Pt`) from 1 to the total number of partitions (`T`).

2. **Identify Cluster Count in Each Partition:**

    - `Find the number of clusters kt in partition Pt`: This line determines the number of clusters (`kt`) in the current base clustering (`Pt`).
3. **Construct Hyperedges:**

    - **Inner loop (`for i = 1 to kt`):** This loop iterates over each cluster (`i`) within the current base clustering (`Pt`).
        - `construct the hyper-edge hi`: This step is not entirely clear in the original code. It's likely that `hi` represents a hyperedge, which should contain information about the cluster being processed (`i`) and potentially the data points within that cluster from `Pt`. The specific construction might involve creating a tuple containing the cluster ID (`i`) and the associated data points, depending on the implementation.
4. **Compute Adjacency Matrix (Optional):**

    - `Compute the adjacency matrix H`: This step calculates the adjacency matrix for the constructed hypergraph. Each element `hij` in the matrix represents the similarity or connection strength between hyperedges `hi` and `hj`. The calculation method might depend on factors like the number of shared data points or co-occurrence across base clusterings. However, the pseudocode doesn't explicitly show how the matrix is populated.
5. **Note:** This step might be optional or implemented differently depending on the chosen consensus function. Some consensus functions might operate directly on

the hypergraph structure without requiring an adjacency matrix.

6. **Determine Maximum Cluster Count (Optional):**

    - `K = max(kt)`: This line (if included) finds the maximum number of clusters (`K`) across all base clusterings. This value might be used by the consensus function (e.g., HMETIS in this case) to guide the partitioning process.
7. **Note:** This step might be unnecessary or calculated differently depending on the chosen consensus function.

8. **Consensus Clustering using HMETIS:**

    - `Pconsensus = HMETIS(H, K)`: This line utilizes the HMETIS algorithm to perform hypergraph partitioning (assuming H is the adjacency matrix and K is the maximum cluster count). The partitioning process aims to identify a set of clusters in the hypergraph that best represent the relationships between clusters across the base clusterings. The output is the final consensus clustering (`Pconsensus`).

Meta-CLustering Algorithm (MCLA)

```
# Input:
#   - partitions: List of lists, where each inner list represents a
cluster partition from a different run (initial clusters)
#   - k: Desired number of meta-clusters

# Output:
#   - final_clusters: List of lists, where each inner list represents a
cluster in the final consensus clustering

# Function to construct the hypergraph
def construct_hypergraph(partitions):
  hypergraph = {}  # Dictionary to store hyperedges (key) and connected
clusters (value)
  for i, partition in enumerate(partitions):
    for j in range(len(partition) - 1):
      for k in range(j + 1, len(partition)):
        cluster1 = partition[j]
        cluster2 = partition[k]
        hyperedge = (i, cluster1, cluster2)  # Tuple representing the
hyperedge
        if hyperedge not in hypergraph:
          hypergraph[hyperedge] = set()
        hypergraph[hyperedge].add(cluster1)  # Add cluster1 to connected
clusters
        hypergraph[hyperedge].add(cluster2)  # Add cluster2 to connected
clusters
  return hypergraph

# Function to calculate similarity between hyperedges
def hyperedge_similarity(hyperedge1, hyperedge2, hypergraph):
```

```
  intersection = len(hypergraph[hyperedge1] & hypergraph[hyperedge2])
  union = len(hypergraph[hyperedge1] | hypergraph[hyperedge2])
  return intersection / float(union)  # Jaccard similarity coefficient

# Function to construct the meta-graph
def construct_meta_graph(hypergraph):
  meta_graph = {}  # Dictionary to store meta-edges and weights
  for hyperedge1 in hypergraph:
    for hyperedge2 in hypergraph:
      if hyperedge1 != hyperedge2:
        similarity = hyperedge_similarity(hyperedge1, hyperedge2,
hypergraph)
        meta_graph[(hyperedge1, hyperedge2)] = similarity
  return meta_graph

# Function to perform meta-clustering using METIS (replace with your
preferred library)
def meta_clustering(meta_graph, k):
  # Assuming METIS takes an adjacency matrix and number of clusters as
input
  adjacency_matrix = [[0 for _ in meta_graph] for _ in meta_graph]  #
Initialize adjacency matrix
  for edge, weight in meta_graph.items():
    i, j = edge
    adjacency_matrix[i][j] = weight
    adjacency_matrix[j][i] = weight  # Add weight for undirected edges
  # Replace with actual METIS implementation
  meta_clusters = METIS(adjacency_matrix, k)  # Cluster the meta-graph
  return meta_clusters

# Function to assign objects (data points) to final clusters
def assign_objects(hypergraph, meta_clusters):
  final_clusters = [[] for _ in range(k)]  # List to store final clusters
  for hyperedge in hypergraph:
    # Find the meta-cluster with the highest association for this
hyperedge
    max_meta_cluster = None
    max_association = 0
    for meta_cluster_id in meta_clusters:
      if hyperedge[0] in meta_clusters[meta_cluster_id]:  # Check if
hyperedge belongs to the meta-cluster
        association = sum(1 for cluster in hypergraph[hyperedge][1:] if
cluster in meta_clusters[meta_cluster_id])
        if association > max_association:
          max_association = association
          max_meta_cluster = meta_cluster_id
    if max_meta_cluster is not None:
```

```
      final_clusters[max_meta_cluster].append(hyperedge[1:])  # Add
associated clusters to final cluster
  return final_clusters

# Main function
def MCLA(partitions, k):
  hypergraph = construct_hypergraph(partitions)
  meta_graph = construct_meta_graph(hypergraph)
  meta_clusters = meta_clustering(meta_graph, k)
  final_clusters = assign_objects(hypergraph, meta_clusters)
  return final_clusters

# Example usage (replace with your actual partitions)
partitions = [
  [1, 2, 3],
  [2, 1, 4],
  [3, 2, 5]
]
k = 2  # Desired number of meta-clusters

final_clusters = MCLA(partitions, k)
```