1) Decision Tree :

ALGORITHM 3.2

```
Input:
   T     //Decision tree
   D     //Input database
Output:
   M     //Model prediction
DTProc algorithm:
            //Simplistic algorithm to illustrate prediction
              technique using DT
   for each t ∈ D do
      n = root node of T;
      while n not leaf node do
         Obtain answer to question on n applied to t;
         Identify arc from t, which contains correct answer;
         n = node at end of this arc;
      Make prediction for t based on labeling of n;
```

2) Naïve Bayes :

Algorithm for Naive Bayes:

Input:

- D: Training dataset with labeled instances.
- X: Unlabeled instances for classification.

Output:

- $P(Y|X)$: Posterior probability of class Y given instance X.

Method:

1. **Training:**
   - Compute prior probabilities $P(Y)$ for each class Y in the training dataset.
   - For each feature X_i and each class Y:
     - Compute the likelihood $P(X_i|Y)$ based on the training data.
     - Use a suitable probability distribution (e.g., Gaussian distribution for continuous features, multinomial distribution for discrete features).
   - Compute the evidence $P(X)$, the probability of observing the features X.

2. **Prediction:**
   - For a given instance X:
     - Compute the posterior probability $P(Y|X)$ for each class Y using Bayes' theorem:
       $P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$
     - Assume feature independence (naive assumption), so $P(X)$ is the same for all classes.
     - The class with the highest posterior probability is the predicted class.
3. **Smoothing (Optional):**
   - To handle the issue of zero probabilities for unseen features, apply smoothing techniques such as Laplace smoothing (additive smoothing).
4. **Output:**
   - $P(Y|X)$: Posterior probability of each class given the instance X.

Note: Naive Bayes is particularly suited for text classification and problems with a large number of features. The choice of probability distribution and smoothing technique depends on the nature of the features in the dataset.

3) K-Means :

ALGORITHM 5.6

```
Input:
   D = {t1, t2, ..., tn}    //Set of elements
   k      //Number of desired clusters
Output:
   K      //Set of clusters
K-means algorithm:
   assign initial values for means m1, m2, ..., mk;
   repeat
      assign each item ti to the cluster which has the closest mean;
      calculate new mean for each cluster;
   until convergence criteria is met;
```

The K-means algorithm is illustrated in Example 5.4.

4) K-Mediods(PAM) :

ALGORITHM 5.8

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A     //Adjacency matrix showing distance between elements
   k     //Number of desired clusters
Output:
   K     //Set of clusters
PAM algorithm:
   arbitrarily select k medoids from D;
   repeat
      for each th not a medoid do
         for each medoid ti do
            calculate TCih;
      find i, h where TCih is the smallest;
      if TCih < 0, then
         replace medoid ti with th;
   until TCih ≥ 0;
   for each ti ∈ D do
      assign ti to Kj, where dis(ti, tj) is the smallest over all medoids;
```

5) Hierarchical Clustering – Single :

ALGORITHM 5.2

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A     //Adjacency matrix showing distance between elements
Output:
   DE    // Dendrogram represented as a set of ordered triples
MST single link algorithm:
   d = 0
   k = n
   K = {{t1}, ..., {tn}}
   DE = (d, k, K); // Initially dendrogram contains each element in
         its own cluster.
   M = MST(A);
   repeat
      oldk = k;
      Ki, Kj = two clusters closest together in MST;
      K = K − {Ki} − {Kj} ∪ {Ki ∪ Kj};
      k = oldk − 1;
      d = dis(Ki, Kj);
      DE = DE ∪ (d, k, K); // New set of clusters added to dendrogram.
      dis(Ki, Kj) = ∞
   until k = 1
```

6) Hierarchical Clustering – Average :

ALGORITHM 5.3

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A     //Adjacency matrix showing distance between elements
Output:
   DE    // Dendrogram represented as a set of ordered triples
Average link algorithm:
   d = 0;
   k = n;
   K = {{t1}, ..., {tn}};
   DE = (d, k, K); // Initially dendrogram contains each element
                   in its own cluster.
   repeat
      oldk = k;
      d = d + 0.5;
      for each pair of Ki, Kj ∈ K do
         ave = average distance between all ti ∈ Ki and tj ∈ Kj;
         if ave ≤ d, then
            K = K − {Ki} − {Kj} ∪ {Ki ∪ Kj};
            k = oldk − 1;
            DE = DE ∪ (d, k, K); // New set of clusters added
                                    to dendrogram.
      until k = 1
```

7) Hierarchical Clustering – Complete :

Algorithm for Hierarchical Clustering (Complete Linkage):

Input:

- X: Set of data points $\{x_1, x_2, \ldots, x_n\}$.
- $\text{distance}(x_i, x_j)$: Distance metric between data points x_i and x_j.
- linkage criterion: Complete Linkage.

Output:

- clusters: Hierarchical clustering structure.

Method:

1. **Initialization:**
   - Start with each data point as a singleton cluster.
2. **Compute Pairwise Distances:**
   - Compute the pairwise distance matrix D based on the specified distance metric.
3. **Merge Closest Clusters:**
   - While more than one cluster exists:
     - Find the two clusters A and B with the smallest pairwise distance according to the chosen linkage criterion (Complete Linkage considers the maximum distance between points in different clusters).
     - Merge clusters A and B into a new cluster C.

4. **Update Distance Matrix:**
   - Update the distance matrix D to reflect the distances between the new cluster C and the remaining clusters.
     $\text{distance}(C, K) = \text{linkage\_criterion}(\text{distance}(A, K), \text{distance}(B, K))$
     for each remaining cluster K.
5. **Repeat Steps 3-4:**
   - Repeat steps 3 and 4 until there is only one cluster.
6. **Output:**
   - The hierarchical clustering structure, often represented as a dendrogram.

Note:

- The linkage criterion determines how the distance between clusters is computed. Complete Linkage, for example, considers the maximum distance between points in different clusters.
- The choice of distance metric and linkage criterion can impact the resulting clustering.

8) Apriori :

ALGORITHM 6.3

```
Input:
   I      //Itemsets
   D      //Database of transactions
   s      //Support
Output:
   L      //Large itemsets
Apriori algorithm:
   k = 0; //k is used as the scan number.
   L = ∅;
```

```
C1 = I;        //Initial candidates are set to be the items.
repeat
   k = k + 1;
   Lk = ∅;
   for each Ii ∈ Ck do
      ci = 0;   // Initial counts for each itemset are 0.
   for each tj ∈ D do
      for each Ii ∈ CK do
         if Ii ∈ tj then
            ci = ci + 1;
   for each Ii ∈ Ck do
      if ci ≥ (s× | D |) do
         Lk = Lk ∪ Ii;
   L = L ∪ Lk;
   Ck+1 = Apriori-Gen(Lk)
until Ck+1 = ∅;
```

9) FP-Tree

Algorithm: FP_growth. Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

1. The FP-tree is constructed in the following steps:
   (a) Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   (b) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following. Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call **insert_tree**$([p|P], T)$, which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call **insert_tree**(P, N) recursively.
2. The FP-tree is mined by calling **FP_growth**(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)    for each combination (denoted as β) of the nodes in the path P
(3)      generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)    generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)    construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)    if Tree_β ≠ ∅ then
(8)      call FP_growth(Tree_β, β); }
```

10) Page Rank :

Algorithm for PageRank:

Input:

- G: Directed graph representing web pages and hyperlinks.
- d: Damping factor (typically set to 0.85).
- ϵ: Convergence threshold for the iterative algorithm.

Output:

- $\mathrm{PageRank}(v)$: PageRank score for each web page v.

Method:

1. **Initialization:**
   - Set an initial PageRank score for each page. Commonly, initialize all scores to a uniform value or use a more sophisticated method.
   - Normalize the scores so that they sum to 1.

2. **Iterative Computation:**
   - Repeat until convergence:
     - For each page v, update its PageRank score based on the scores of pages linking to it:
       $$\text{PageRank}(v) = (1-d) + d \times \left(\frac{\text{PageRank}(u)}{L(u)} + \frac{\text{PageRank}(w)}{L(w)} + \dots \right)$$
       where $u, w, \dots$ are pages linking to v, and $L(u), L(w), \dots$ are the number of outgoing links from $u, w, \dots$.
     - Check for convergence using a threshold ϵ.
3. **Output:**
   - The final PageRank scores for each web page.

Note:

- The damping factor (d) is introduced to model the behavior of a random web surfer who, with probability $1 - d$, jumps to a random page, and with probability d, follows one of the outgoing links.
- The algorithm continues iterating until the PageRank scores converge or a predefined maximum number of iterations is reached.
- The choice of initialization method can affect convergence speed, and the damping factor is often set empirically.

11) HITS

ALGORITHM 7.1

```
Input:
   W          //WWW viewed as a directed graph
   q          //Query
   s          //Support
Output:
   A          //Set of authority pages
   H          //Set of hub pages
HITS algorithm
   R = SE(W, q)
   B = R ∪ {pages linked to from R} ∪ {pages that link to pages in R};
   G(B, L) =  Subgraph of W induced by B;
   G(B, L^1) =  Delete links in G within same site;
   x_p = Σ_q where (q,p)∈L^1 y_q;     // Find authority weights;
   y_p = Σ_q where (p,q)∈L^1 x_q;     // Find hub weights;
   A = {p | p has one of the highest x_p};
   H = {p | p has one of the highest y_p};
```

For Information package and Designing schema refer Exp. 2

For OLAP operations refer Exp. 7