Lab – Catch-up / Clustering #2

Data Mining, Spring 2018

Carolina (berm@itu.dk) | Daniel (dafr@itu.dk) | Mathias (stsa@itu.dk)

IT UNIVERSITY OF COPENHAGEN

Today's Lab: Catch-up and optional exercises

Catchup Lab

- Today's lab is mostly a catch up lab
  - Finish the previous labs
  - Or work on your individual assignment due on April 2nd.
  - Walk through of pseudo code
- Optional exercises:
  - Implement the GSP (Generalized Sequential Patterns) algorithm
  - Implement the FSG (Frequent Subgraph) algorithm

Preprocessing

Catch-up Lab

- Normalization
- Cleaning data
- Transforming data
- Data reduction

Normalization

Normalization

- **Min-max normalization**: to [new_min$_A$, new_max$_A$]

$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

  - Ex. Let income range \$12,000 to \$98,000 normalized to [0.0, 1.0]. Then \$73,000 is mapped to $\frac{73{,}600-12{,}000}{98{,}000-12{,}000}(1.0-0)+0=0.716$

- **Z-score normalization** (μ: mean, σ: standard deviation):

$$v' = \frac{v - \mu_A}{\sigma_A}$$

  - Ex. Let μ = 54,000, σ = 16,000. Then $\frac{73{,}600-54{,}000}{16{,}000}=1.225$

- **Normalization by decimal scaling**

$$v' = \frac{v}{10^j}$$ Where j is the smallest integer such that Max(|v'|) < 1

Sample Code: Min-max

```java
/**
 * Normalize x.
 * @param x The value to be normalized.
 * @return The result of the normalization.
 */
public double normalize(double x) {
    return ((x - dataLow)
            / (dataHigh - dataLow))
            * (normalizedHigh - normalizedLow) + normalizedLow;
}
```

```java
public static double arrayMax(double[] arr) {
    double max = Double.NEGATIVE_INFINITY;

    for(double cur: arr)
        max = Math.max(max, cur);

    return max;
}
```

You can find this and other examples on finding the max and min value in an array in Java here.

Decimal scaling

- **Normalization by decimal scaling**

$$v' = \frac{v}{10^j}$$ Where j is the smallest integer such that Max($|v'|$) < 1

Decimal Scaling Normalization

Suppose that the recorded values of F range from -986 to 917. The maximum absolute value of F is 986. To normalize by decimal scaling, we therefore divide each value by 1,000 (i.e., $j = 3$) so that -986 normalizes to -0.986 and 917 normalizes to 0.917.

Classification

k-NN

- Step 1: Determine parameter K = number of nearest neighbors
- Step 2: Calculate the distance between the query-instance and all the training examples.
- Step 3: Sort the distance and determine nearest neighbors based on the k-th minimum distance.
- Step 4:Gather the category Y of the nearest neighbors.
- Step 5: Use simple majority of the category of nearest neighbors as the prediction value of the query instance.

In this link you can find examples of KNN using different distances.
In this article, you can find implementations and discussions of the different distances.

ID3

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C, then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let D_j be the set of data tuples in D satisfying outcome j; // a partition
(12)     if D_j is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(D_j, attribute_list) to node N;
     endfor
(15) return N;
```

Pseudocode explained with examples here and here

ID3 (Learning Sets S, Attributes Sets A, Attributesvalues V) Return Decision Tree.

Begin

Load learning sets first, create decision tree root node 'rootNode', add learning set S into root node as its subset.

For rootNode, we compute Entropy(rootNode.subset) first

If Entropy(rootNode.subset)==0, then rootNode.subset consists of records all with the same value for the categorical attribute, return a leaf node with **decision attribute:attribute value;**

If Entropy(rootNode.subset)!=0, then compute information gain for each attribute left(have not been used in splitting), find attribute A with Maximum(Gain(S,A)). Create child nodes of this rootNode and add to rootNode in the decision tree.

For each child of the rootNode, apply ID3(S,A,V) recursively until reach node that has entropy=0 or reach leaf node.

End ID3.

ID3: formulas

Entrophy(S) = ∑ – p(I) . log2p(I)

Gain(S, A) = Entrophy(S) – ∑ [p(S|A) .Entrophy(S|A)]

ID3: example - decision making factors to play tennis at outside

Day	Outlook	Temp.	Humidity	Wind	Decision
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

ID3: example - entrophy

Find the example here

Initially, we need to calculate the overall entrophy.

Decision column consists of 14 instances and includes two labels: yes and no. There are 9 decisions labeled yes, and 5 decisions labeled no.

$$\text{Entrophy(Decision)} = -\ p(\text{Yes}) \cdot \log_2 p(\text{Yes}) - p(\text{No}) \cdot \log_2 p(\text{No})$$

$$\text{Entrophy(Decision)} = -\ (9/14) \cdot \log_2(9/14) - (5/14) \cdot \log_2(5/14) = 0.940$$

Now, we need to find the most dominant factor for decisioning.

ID3: example of formulas – wind factor on decision

Gain(Decision, Wind) = Entrophy(Decision) – ∑ [p(Decision|Wind) . Entrophy(Decision|Wind)]

Wind attribute has two labels: weak and strong. We would reflect it to the formula.

Gain(Decision, Wind) = Entrophy(Decision) – [p(Decision|Wind=Weak) . Entrophy(Decision|Wind=Weak)] – [p(Decision|Wind=Strong) . Entrophy(Decision|Wind=Strong)]

We need to calculate (Decision|Wind=Weak) and (Decision|Wind=Strong) respectively.

There are 8 instances for weak wind. Decision of 2 items are no, whereas 6 items are yes as illustrated below.

Entrophy(Decision|Wind=Weak) = – p(No) . $\log_2$p(No) – p(Yes) . $\log_2$p(Yes)

Entrophy(Decision|Wind=Weak) = – (2/8) . $\log_2$(2/8) – (6/8) . $\log_2$(6/8) = 0.811

ID3: example of formulas – wind factor on decision

There are 6 instances for strong wind. Decision is divided into two equal parts.

Entrophy(Decision|Wind=Strong) = – p(No) . $\log_2$p(No) – p(Yes) . $\log_2$p(Yes)

Entrophy(Decision|Wind=Strong) = – (3/6) . $\log_2$(3/6) – (3/6) . $\log_2$(3/6) = 1

Now, we can turn back to Gain(Decision, Wind) equation.

Gain(Decision, Wind) = Entrophy(Decision) – [p(Decision|Wind=Weak) . Entrophy(Decision|Wind=Weak)] – [p(Decision|Wind=Strong) . Entrophy(Decision|Wind=Strong)]

Gain(Decision, Wind) = 0.940 – [(8/14) . 0.811] – [(6/14). 1] = 0.048

Calculations for wind column is over. Now, we need to apply same calculations for other columns to find the most dominant factor on decision.

ID3: from decision tree to decision rules

A decision tree can easily be transformed to a set of rules by mapping from the root node to the leaf nodes one by one.

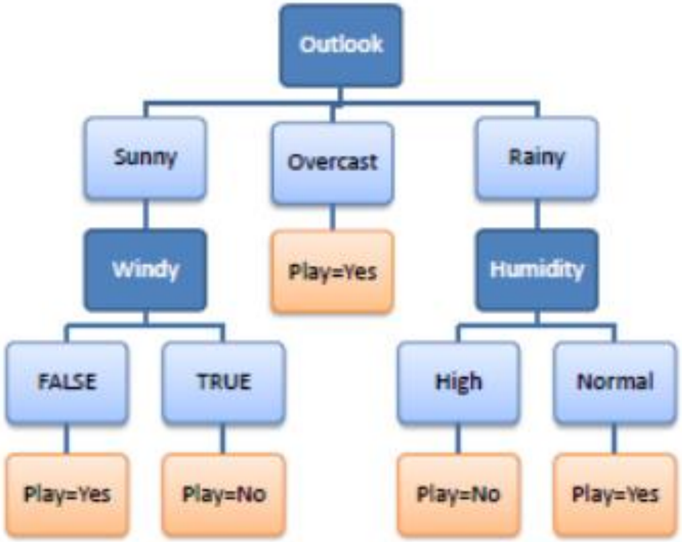
R_1: IF (Outlook=Sunny) AND (Windy=FALSE) THEN Play=Yes

R_2: IF (Outlook=Sunny) AND (Windy=TRUE) THEN Play=No

R_3: IF (Outlook=Overcast) THEN Play=Yes

R_4: IF (Outlook=Rainy) AND (Humidity=High) THEN Play=No

R_5: IF (Outlook=Rain) AND (Humidity=Normal) THEN Play=Yes



Pattern Mining

Apriori

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

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

Output: L, frequent itemsets in D.

Method:

```
L_1 = find_frequent_1-itemsets(D);
for (k = 2; L_{k-1} ≠ φ; k++) {
    C_k = apriori_gen(L_{k-1});
    for each transaction t ∈ D { // scan D for counts
        C_t = subset(C_k, t); // get the subsets of t that are candidates
        for each candidate c ∈ C_t
            c.count++;
    }
    L_k = {c ∈ C_k | c.count ≥ min_sup}
}
return L = ∪_k L_k;

procedure apriori_gen(L_{k-1}:frequent (k − 1)-itemsets)
    for each itemset l_1 ∈ L_{k-1}
        for each itemset l_2 ∈ L_{k-1}
            if (l_1[1] = l_2[1]) ∧ (l_1[2] = l_2[2])
                ∧ ... ∧ (l_1[k−2] = l_2[k−2]) ∧ (l_1[k−1] < l_2[k−1]) then {
                c = l_1 ⋈ l_2; // join step: generate candidates
                if has_infrequent_subset(c, L_{k-1}) then
                    delete c; // prune step: remove unfruitful candidate
                else add c to C_k;
            }
    return C_k;

procedure has_infrequent_subset(c: candidate k-itemset;
        L_{k-1}: frequent (k − 1)-itemsets); // use prior knowledge
    for each (k − 1)-subset s of c
        if s ∉ L_{k-1} then
            return TRUE;
    return FALSE;
```

Apriori - example

Minimum Support: 2

1

Tid	Items Bought
1	Milk, Tea, Cake
2	Eggs, Tea, Cold Drink
3	Milk, Eggs, Tea, Cold Drink
4	Eggs, Cold Drink
5	Juice

2

Items Bought	Support
Milk	2
Eggs	3
Tea	3
Cold Drink	3
Juice	1
Cake	1

3

Items Bought	Support
Milk	2
Eggs	3
Tea	3
Cold Drink	3

4

Items Bought
Milk, Eggs
Milk, Tea
Milk, Cold Drink
Eggs, Tea
Eggs, Cold Drink
Tea, Cold Drink

5

Items Bought	Support
Milk, Eggs	1
Milk, Tea	2
Milk, Cold Drink	1
Eggs, Tea	2
Eggs, Cold Drink	3
Tea, Cold Drink	2

6

Items Bought	Support
Milk, Tea	2
Eggs, Tea	2
Eggs, Cold Drink	3
Tea, Cold Drink	2

7

Items Bought	Support
Eggs, Tea, Cold Drink	2

There is only one itemset with minimum support 2.
So only one itemset is frequent

Step 1: Data in the database

[quads id=1]

Step 2: Calculate the support/frequency of all items

Step 3: Discard the items with minimum support less than 2

Step 4: Combine two items

Step 5: Calculate the support/frequency of all items

Step 6: Discard the items with minimum support less than 2

Step 6.5: Combine three items and calculate their support.

Step 7: Discard the items with minimum support less than 2

Result:

Only one itemset is frequent (Eggs, Tea, Cold Drink) because this itemset has minimum support 2

Source [link]

Clustering

k-means

Algorithm: *k*-means. The *k*-means algorithm for partitioning, where each cluster's center is represented by the mean value of the objects in the cluster.

Input:

- *k*: the number of clusters,
- *D*: a data set containing *n* objects.

Output: A set of *k* clusters.

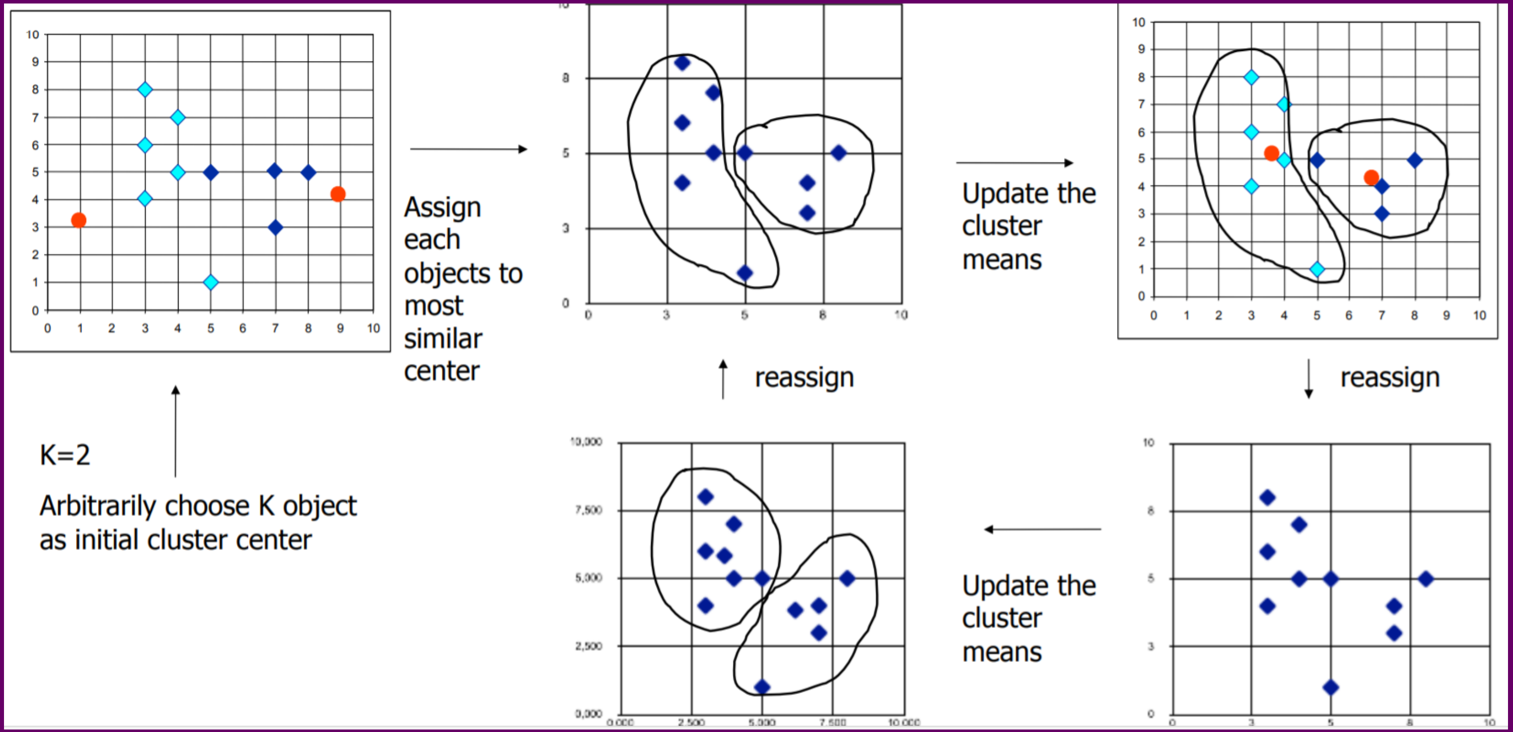
Method:

(1) arbitrarily choose *k* objects from *D* as the initial cluster centers;
(2) **repeat**
(3)   (re)assign each object to the cluster to which the object is the most similar, based on the mean value of the objects in the cluster;
(4)   update the cluster means, that is, calculate the mean value of the objects for each cluster;
(5) **until** no change;

Remember to normalize each dimension first

Example of the implementation with 2 dimensions

k-means



Example from the lecture

k-medoids

Algorithm: ***k*-medoids.** PAM, a k-medoids algorithm for partitioning based on medoid or central objects.

Input:

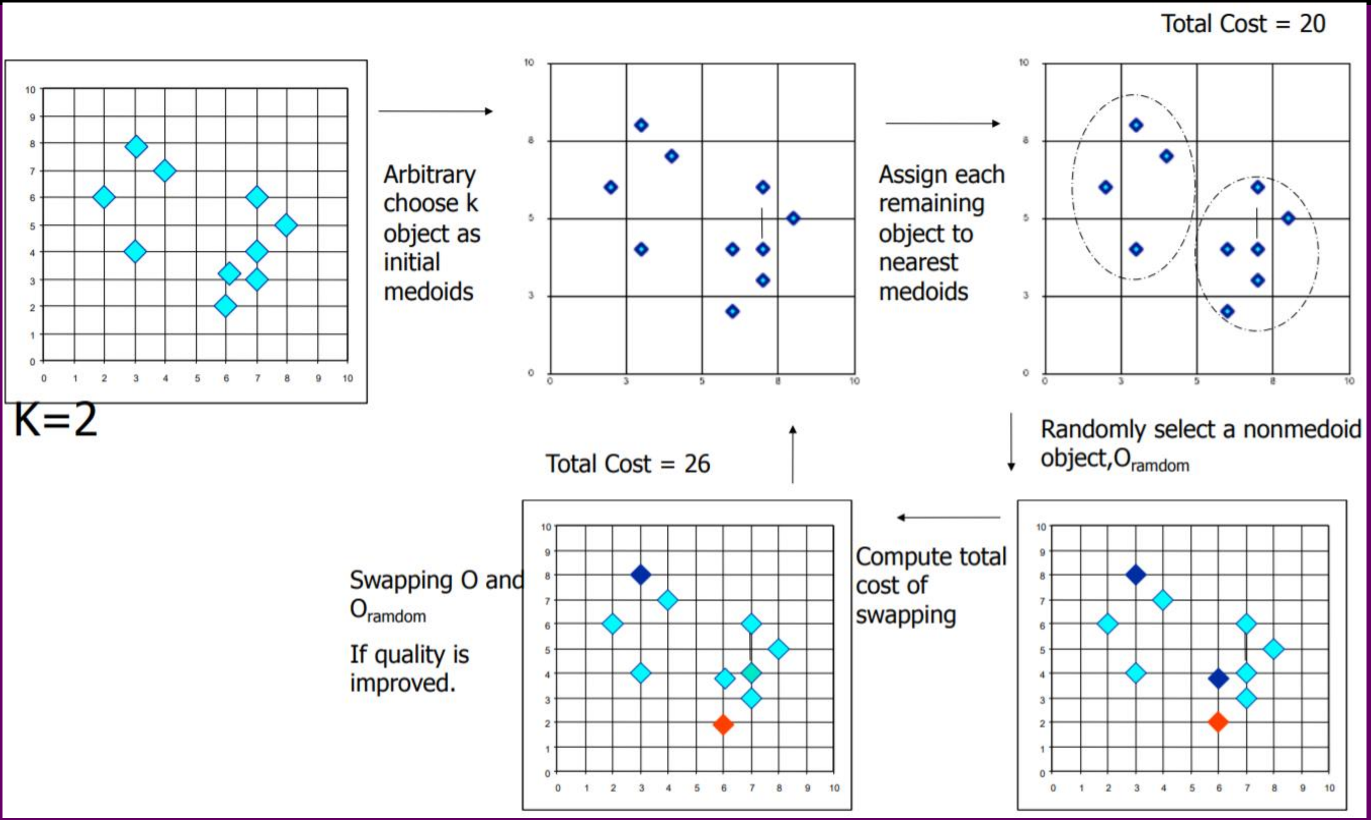
- k: the number of clusters,
- D: a data set containing n objects.

Output: A set of k clusters.

Method:

(1) arbitrarily choose k objects in D as the initial representative objects or seeds;
(2) **repeat**
(3) assign each remaining object to the cluster with the nearest representative object;
(4) randomly select a nonrepresentative object, $\boldsymbol{o}_{random}$;
(5) compute the total cost, S, of swapping representative object, $\boldsymbol{o}_j$, with $\boldsymbol{o}_{random}$;
(6) **if** $S < 0$ **then** swap $\boldsymbol{o}_j$ with $\boldsymbol{o}_{random}$ to form the new set of k representative objects;
(7) **until** no change;

k-medoids



Example from the lecture

Thanks for listening!