Chapter 10

Unsupervised Learning

10.1 Introduction

Two basic approaches can be distinguished in this area: *finding regularities in data* (discovery) and *conceptual clustering*. The former approach is basically connected with the famous AM program [Davis and Lenat,1982], designed originally to discover concepts in mathematics. AM uses some notions of set theory, operations for creating new concepts by modifying and combining existing ones, and a set of heuristics for detecting "interesting" concepts. For example, AM discovered the natural numbers by modifying its notion of multiset (set with multiple occurrences of the same element). By using multiset of a single element AM discovered a representation of natural numbers as multisets and the operation on numbers as set operations (e.g. $\{1,1,1\} \cup \{1,1\} = \{1,1,1,1,1\}$ corresponds to $3+2=5$). The heuristics which AM used to evaluate the interesting concepts were very domain dependent. Therefore it was difficult to apply the system in other fields beyond the elementary number theory.

Clustering is known from mathematics, where the basic idea is to group some object in a cluster using the euclidean distance between them as a criterion for their "similarity". When using structural description of the objects, however, the traditional clustering algorithms fail to capture any domain knowledge related to the object descriptions. Another disadvantage is that these algorithms represent clusters *extensionally*, i.e. by enumerating all their members. However, an *intensional* description of the clusters (i.e. such that can be used for classifying objects by using their descriptions in terms of relations, properties, features etc.) can produce a semantic explanation of the resulting categories, which is more human-like.

Conceptual clustering is an approach, which addresses the above mentioned problems. We shall discuss briefly two instances of this approach - CLUSTER/2 [Michalski and Stepp, 1983] and COBWEB [Gennari et al, 1989].

10.2 CLUSTER/2

This algorithm forms k categories by constructing individual objects grouped around k *seed* objects. It is as follows:

1. Select k objects (seeds) from the set of observed objects (randomly or using some selection function).

2. For each seed, using it as a positive example the all the other seeds as negative examples, find a maximally general description that covers all positive and none of the negative examples.

3. Classify all objects form the sample in categories according to these descriptions. Then replace each maximally general descriptions with a maximally specific one, that cover all objects in the category. (This possibly avoids category overlapping.)

4. If there are still overlapping categories, then using some metric (e.g. euclidean distance) find central objects in each category and repeat steps 1-3 using these objects as seeds.

5. Stop when some quality criterion for the category descriptions is satisfied. Such a criterion might be the complexity of the descriptions (e.g. the number of conjuncts)

6. If there is no improvement of the categories after several steps, then choose new seeds using another criterion (e.g. the objects near the edge of the category).

The underlying idea of the above algorithm is to find *necessary and sufficient conditions for category membership*. There is however some psychological evidence that human categorization is based on the notion of *prototypicality*. For example, the *family resemblance theory* [Wittgenstein, 1953] argues that categories are defined by a system of similarities between the individual members of a category.

Another feature of human categorization is the use of *base-level* categories. In contrast to the formal hierarchies used often in AI (e.g. the taxonomic trees, Chapter 1), the humans mostly use categories which are neither most general, nor most specific. For example, the concept of "chair" is most basic than both its generalization "furniture" and its specialization "office chair", and "car" is more basic than both "porshe" and "vehicle".

The COBWEB algorithm, though not designed as a cognitive model, accounts for the above mentioned features of human categorization.

10.3 COBWEB

COBWEB is an incremental learning algorithm, which builds a taxonomy of categories without having a predefined number of categories. The categories are represented *probabilistically* by the conditional probability $p(f_i = v_{ij}|c_k)$ with which feature f_i has value v_{ij}, given that an object is in category c_k.

Given an instance COBWEB evaluates the quality of either placing the instance in an existing category or modifying the hierarchy to accommodate the instance. The criterion used for this evaluation is based on *category utility*, a measure that Gluck and Corter have shown predicts the basic level ound in psychological experiments. Category utility attempts to maximize both the probability that two objects in the same category have values in common and the probability that objects from different categories have different feature values. It is defined as follows:

$$\sum_k \sum_i \sum_j p(f_i = v_{ij}) p(f_i = v_{ij}|c_k) p(c_k|f_i = v_{ij})$$

The sum is calculates for all categories, all features and all values. $p(f_i = v_{ij}|c_k)$ is called *predictability*, i.e. this is the probability that an object has value v_{ij} for its feature f_j, given that it belongs to category c_k. The higher this probability, the more likely two objects in a category share the same feature values. $p(c_k|f_i = v_{ij})$ is called *predictiveness*, i.e. the probability that an object belongs to category c_k, given that it has value v_{ij} for its feature f_i. The greater this probability, the less likely objects from different categories will have feature values in common. $p(f_i = v_{ij})$ is a weight, assuring that frequently occurring feature values will have stronger influence on the evaluation.

Using the Bayes' rule we have $p(a_i = v_{ij})p(c_k|a_i = v_{ij}) = p(c_k)p(a_i = v_{ij}|c_k)$. Thus we can transform the above expression into an equivalent form:

$$\sum_k p(c_k) \sum_i \sum_j p(f_i = v_{ij}|c_k)^2$$

Gluck and Corter have shown that the subexpression $\sum_i \sum_j p(f_i = v_{ij}|c_k)^2$ is the *expected* number of attribute values that one can correctly guess for an arbitrary member of class c_k. This expectation assumes a *probability matching* strategy, in which one guesses an attribute value with a probability equal to its probability of occurring. They define the category utility as the *increase* in the expected number of attribute values that can be correctly guessed, given a set of n categories, over the expected number of correct guesses without such knowledge. The latter term is $\sum_i \sum_j p(f_i = v_{ij})^2$, which is to be subtracted from the above expression. Thus the complete expression for the category utility is the following:

$$\frac{\sum_k p(c_k) \sum_i \sum_j [p(f_i = v_{ij}|c_k)^2 - p(f_i = v_{ij})^2]}{k}$$

The difference between the two expected numbers is divided by k, which allows us to compare different size clusterings.

When a new instance is processed the COBWEB algorithm uses the discuss above measure to evaluate the possible clusterings obtained by the following actions:

– classifying the instance into an existing class;
– creating a new class and placing the instance into it;
– combining two classes into a single class (merging);
– dividing a class into two classes (splitting).

Thus the algorithm is a *hill climbing* search in the space of possible clusterings using the above four operators chosen by using the category utility as an evaluation function.

cobweb($Node$,$Instance$)
begin

- If $Node$ is a leaf then begin
  Create two children of Node - L_1 and L_2;
  Set the probabilities of L_1 to those of $Node$;
  Set the probabilities of L_2 to those of $Insnatce$;
  Add $Instance$ to $Node$, updating $Node$'s probabilities.
  end

- else begin
  Add $Instance$ to $Node$, updating $Node$'s probabilities; For each child C of $Node$, compute the category utility of clustering achieved by placing $Instance$ in C;
  Calculate:
  S_1 = the score for the best categorization ($Instance$ is placed in C_1);
  S_2 = the score for the second best categorization ($Instance$ is placed in C_2);
  S_3 = the score for placing $Instance$ in a new category;
  S_4 = the score for merging C_1 and C_2 into one category;
  S_5 = the score for splitting C_1 (replacing it with its child categories.
  end

- If S_1 is the best score then call cobweb(C_1, $Instance$).

- If S_3 is the best score then set the new category's probabilities to those of $Instance$.

- If S_4 is the best score then call cobweb(C_m, $Instance$), where C_m is the result of merging C_1 and C_2.

- If S_5 is the best score then split C_1 and call cobweb($Node$, $Instance$).

end

Consider the following set of instances. Each one defines a one-celled organism using the values of three features: *number of tails*, *color* and *number of nucleis* (the first element of the list is the name of the instance):

```
instance([cell1,one,light,one]).
instance([cell2,two,dark,two]).
instance([cell3,two,light,two]).
instance([cell4,one,dark,three]).
```

The following is a trace of a program written in Prolog implementing the COBWEB algorithm:

```
 Processing instance cell1 ...
 Root initialized with instance: node(root,1,1)

 Processing instance cell2 ...
 Root node:node(root,1,1) used as new terminal node:node(node_1,1,1)
 Case cell2 becomes new terminal node(node_2,1,1)
 Root changed to: node(root,2,0.5)

 Processing instance cell3 ...
 Root changed to: node(root,3,0.555553)
 Incorporating instance cell3 into node: node(node_4,2,0.833333)
 Using old node: node(node_2,1,1) as terminal node.
 New terminal node: node(node_7,1,1)

 Processing instance cell4 ...
 Root changed to: node(root,4,0.458333)
 New terminal node: node(node_10,1,1)
yes

?- print_kb.
d_sub(root,node_10).
d_sub(node_4,node_7).
d_sub(node_4,node_2).
d_sub(root,node_4).
d_sub(root,node_1).

node_10(nuclei,[three-1]).
node_10(color,[dark-1]).
node_10(tails,[one-1]).
root(nuclei,[one-1,three-1,two-2]).
root(color,[dark-2,light-2]).
root(tails,[one-2,two-2]).
node_7(nuclei,[two-1]).
node_7(color,[light-1]).
node_7(tails,[two-1]).
node_4(nuclei,[two-2]).
node_4(color,[dark-1,light-1]).
node_4(tails,[two-2]).
node_2(nuclei,[two-1]).
```

```
node_2(color,[dark-1]).
node_2(tails,[two-1]).
node_1(tails,[one-1]).
node_1(color,[light-1]).
node_1(nuclei,[one-1]).
```

The *print_kb* predicate prints the category hierarchy (*d_sub* structures) and the description of each category (*node_i* structures). The first argument of the *node_i* structures is the corresponding feature, and the second one is a list of pairs $Val - Count$, where $Count$ is a number indicating the number of occurrences of Val as a value of the feature.