That is, it simply memorizes the training set. As a consequence, it is unable to decide upon the class of the unseen instances, even while it is flawless on the training set itself. This is an example of overfitting. It is unable to generalize from the examples.

Given a hypothesis space H, a hypothesis $h \in H$ is said to *overfit the* training data, if there exists some alternative hypothesis $h' \in H$, such that h has a smaller error over the training examples, but h' has a smaller error than h over the entire distribution of instances (Mitchell, 1997).

On the other hand, with the conjunctive hypotheses schema the system is learning from training data and is able to generalize. But there is a caveat, the inductive bias. The bias is that the system works only when the target function is contained in the space of conjunctive hypotheses. In a sense, the bias in the system is able to learn and generalize, only when there is a restricted form for the hypotheses which it tunes to the training examples.

The fundamental property of inductive learning is that a learner that makes no *a priori* assumptions regarding the identity of the target concept has no rational basis for classifying unseen instances (Mitchell, 1997).

Because inductive learning requires some form of prior assumption (expectation) or inductive bias, we will find it useful to characterise different learning approaches by the inductive bias they employ.

Does that mean that it is not possible to learn more complex target concepts? No, one can, but with a different kind of bias, as we see in the next section.

18.5 Decision Trees

Decision trees, also known as discrimination trees, or many sorted trees adopt a different approach to concept learning. Instead of trying to learn a specific concept, they accept a training set labelled with different class labels (concept names) and build a discrimination structure that separates the different classes or concepts.

Given that the elements of the domain are described by attributes, the objective is to uncover what combination of attributes defines a given concept. Another way of looking at this is to ask which attributes are characteristic of a given class, and which discriminate it from other classes. Decision trees, when constructed, ask a series of questions of the given new instance to be classified, and the answer to each question leads to traversal down the corresponding branch, eventually culminating in a class label. The process is comparable to the manner in which a physician asks a patient questions, where the next question depends upon the previous answer, leading to a diagnosis.

We often make these kinds of generalizations implicitly. A teacher may believe that a student with high attendance and serious countenance is a good student, and a tennis player may conclude that a sunny day with normal humidity is perfectly suited to play tennis. While we arrive at

these conclusions by a process of accumulated experience, decision tree building algorithms require that all the data be available along with their class labels. That is, like the algorithms *Find-S* and *Candidate-Elimination* seen earlier, it is a *supervised learning* procedure.

Unlike the other two algorithms though, decision tree building algorithms can tolerate erroneous data or noise, and one can also control the level of detail at which discrimination should be done. These algorithms explore the space of decision trees and they have a preference bias for smaller trees. In fact by controlling the tree construction process, one can check the phenomenon of overfitting.

We look at the well known ID3 (Iterative Dichotomiser 3) algorithm devised by Ross Quinlan (1986). Given a training set with N elements from K classes, the basic idea is to identify those attributes whose values separate the different classes. Each such attribute then becomes a question to ask of a data set, and partitions the data set based on the different values. The idea then, like in the game of Twenty Questions[5], is to identify the class in as few questions as possible.

The algorithm associates the entire data set with the root node of the decision tree that it is constructing. The data set may have elements of different classes. The task to choose an attribute whose answer will separate the classes as much as possible. The different partitions are then treated recursively in the same manner, till the partitions have elements of only one class or some other termination criterion is used. One can now imagine why it can tolerate errors. If an element with a wrong class label has crept into the data set, it will go along with the elements of the "correct" class. If the process is continued till the partitions are completely homogenous then this element would have separated towards the end, and can be identified as being wrongly labelled. Some amount of post-processing can then prune the tree of such tiny spurious classes.

Different attributes will induce different partitions on a given data set. How does one choose the attribute that will separate the different classes most? The approach used in many algorithms is to use the notion of *entropy*, which is a measure of information content in a set (Shannon, 1948). Entropy is a measure of diversity in a set. The more homogenous the set is, the less information it has; in the sense that it needs a smaller number of bits to describe it. The more heterogeneous it is, the greater the information content or entropy. Entropy is then a measure of predictability. If one were to choose a random element from the set then zero entropy would mean that it is entirely predictable. This would happen if all the elements in the set were of the same class. With equal elements from two classes, the entropy would be one. With more classes, the entropy goes up further.

Given a set S of elements from K classes, the entropy of the set is defined as,

$$Entropy(S) = \sum_{i=1}^{K} -p_i * \log_2 p_i \tag{18.45}$$

where p_i is the proportion of elements of the i^{th} class in the set. The algorithm tries out partitioning the set of element using different attributes. Let A be an attribute and $Values(A)$ the set of values for the attribute. Using the attribute A to partition the set S results in information gain $Gain(S, A)$, as defined by the formula below.

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} * Entropy(S_v) \tag{18.46}$$

The information gain is computed by subtracting the weighted sum of the entropies of the partitions from the entropy of the original set S. The weight for each partition is the proportion of the elements in that partition. One can see that the more homogenous the partitions are, the greater is the information gain. This is illustrated in Figure 18.10. The algorithm tries out the different attributes available and picks the one that yields the greatest information gain. The process continues recursively on the partitions.

Let us look at this process for the training set given in Table 18.3. There are 48 employees in the training set S of which there are 11 with the *High* (salary) label, 16 with the *Low* label, 8 with *Medium*, and 13 with the *VeryHigh* label, in alphabetical order of the labels. The entropy of the set is computed as follows.

$$\begin{aligned}
\text{Entropy}(S) = & -(11/48) * \log_2(11/48) \\
& -(16/48) * \log_2(16/48) \\
& -(8/48) * \log_2(8/48) \\
& -(13/48) * \log_2(13/48) \\
= & -0.229 * -2.126 \\
& -0.333 * -1.585 \\
& -0.167 * -2.585 \\
& -0.271 * -1.885 \\
= & \ 0.487 + 0.528 + 0.431 + 0.510 \\
= & \ 1.957
\end{aligned}$$

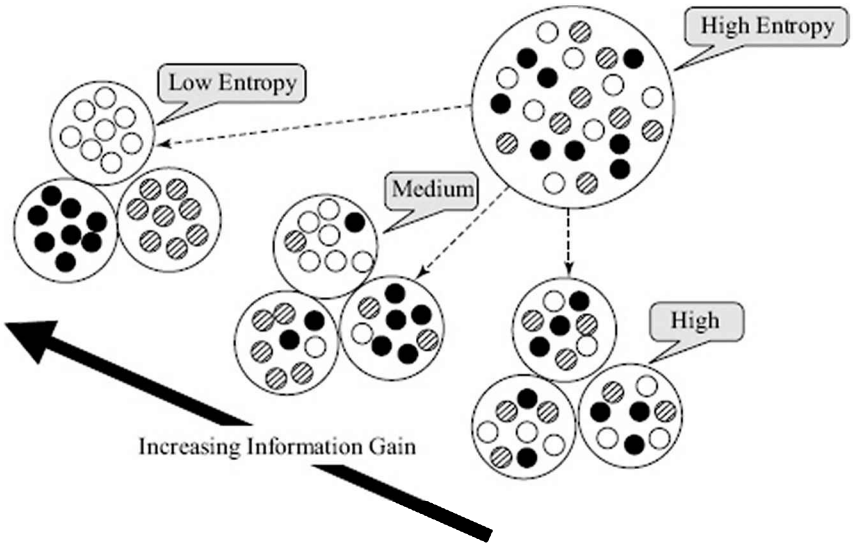

FIGURE 18.10 Partitioning a set into subsets could result in information gain, if the subsets have lower entropy. The algorithm to construct decision trees pick that attribute to partition the set which results in highest information gain. In the figure, three illustrative choices are shown.

The set *S* can be partitioned by the three attributes: *Exp-Symbol* (the nominal version of the numeric attribute *Experience*), *Education* and *Hands-on*. The first attribute *Exp-Symb* produces three partitions for the three values *High*, *Low* and M*edium*. *Education* also produces three partitions for the values B*achelor* or *B*, *Master* or *M*, and *PhD* or *P*, while *Hands-On* produces two partitions. The possible partitions and the number of elements in them are shown in Table 18.4.

Table 18.4 The distribution of elements by the three attributes Exp-Symb, Education, and HandOn

S		*Exp-Symb*			*Education*			*HandOn*	
Class		*High*	*Low*	*Medium*	*B*	*M*	*P*	*No*	*Yes*
High	11	**6**	**0**	**5**	**5**	**4**	**2**	**6**	**5**
Low	16	0	12	4	10	3	3	13	3
Medium	6	**0**	**3**	**5**	**0**	**6**	**2**	**5**	**3**
Very High	13	7	2	4	2	3	8	0	13
Total	48	**13**	**17**	**18**	**17**	**16**	**15**	**24**	**24**

For example, the $High_{Exp}$ partition induced by *Exp-symb* has 13 elements of which 6 have the label *High* (salary) and 7 have label *VeryHigh*. The entropy of this set is,

$$\begin{aligned} Entropy(High_{Exp}) &= -(6/13)*\log_2(6/13) - (7/13)*\log_2(7/13) \\ &= -0.462 * -1.115 - 0.538 * -0.893 \\ &= 0.515 + 0.481 \\ &= 0.996 \end{aligned}$$

Observe that this has only two terms because there are elements of only two classes in the partition. The entropies of the other two partitions are computed similarly.

$$Entropy(Low_{Exp}) = 1.160$$
$$Entropy(Medium_{Exp}) = 1.991$$

The information gain for *Exp-Symb* is computed as shown.

$$\begin{aligned} Gain(S, Exp\text{-}Symb) &= Entropy(S) \\ &\quad -(|High_{Exp}|/|Exp\text{-}Symb|) * Entropy(High_{Exp}) \\ &\quad -(|Low_{Exp}|/|Exp\text{-}Symb|) * Entropy(Low_{Exp}) \\ &\quad -(|Medium_{Exp}|/|Exp\text{-}Symb|) * Entropy(Medium_{Exp}) \\ &= 1.957 \\ &\quad -0.271 * 0.996 \\ &\quad -0.354 * 1.160 \\ &\quad -0.375 * 1.991 \\ &= 0.530 \end{aligned}$$

Similarly, the gain for the other two attributes are,

$Gain(S, Education) = 0.301$

$Gain(S. Hands\text{-}On) = 0.381$

The attribute *Exp-Symb* yields the maximum information gain and is therefore used to partition the set *S* into three partitions $High_{Exp}$ with 13 elements, Low_{Exp} with 17 elements, and $Medium_{Exp}$ with 18 of the original 48 elements. The process continues recursively with these three sets. The resulting decision tree is shown in Figure 18.11.

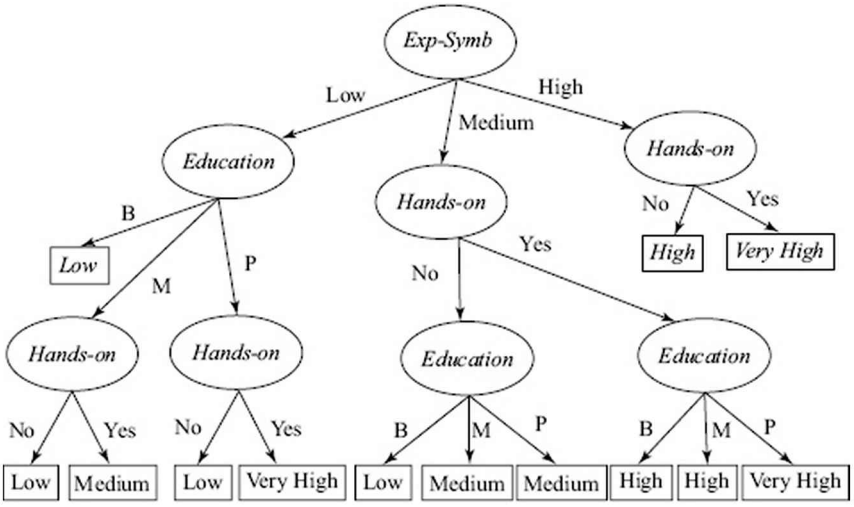

FIGURE 18.11 The decision tree produced by the algorithm ID3 for the data set in Table 18.3.

The decision tree is basically a discriminative structure. A new instance can be "inserted" in the root and it traverses down the tree eventually falling into a bucket at the leaf node, labelled with its class name. We have seen variations on the theme in the *Rete net* in Chapter 6 and the *kd-tree* in Chapter 15.

Nevertheless, a concept description can be constructed by inspecting the tree. Every path from the root to the concept is a conjunction of constraints on a set of attributes on the path. The different paths together are different sets of alternative sets of constraints, and can be joined together in a disjunction. Let us look at the concept *VeryHigh* in our training set. The leftmost path in the figure represents the constraints <*Exp-Symb=Low*, *Education=PhD*, *HandsOn=Yes*> which we can write as <L,P,Y>. The second one represents <*Exp-Symb=Medium*, *HandsOn=Yes*, *Education=PhD*> which in the order of our schema is <M,P,Y>. The rightmost path represents the constraints <*Exp-Symb=High*, *HandsOn=Yes*>. Since it does not say anything about *Education*, we can assume that any value would do and we can represent the constraints as <H,?,Y>.

Thus, the hypothesis that matches the training set is,

$$<Low,\ P,\ Yes> \vee <Medium,\ P,\ Yes> \vee <High,\ ?,\ Yes>$$

as found by the ID3 algorithm. The reader is encouraged to verify that this can be simplified to <?, *P*, *Yes*> ˅ <*High*, ?, *Yes*>. This essentially says that to be in the *VeryHigh* salary category, one needs to be *Hands-On* and in addition either have a *PhD* degree or *High* levels of experience.

The question is how does ID3 algorithm working with a complete hypotheses space produce a structure that can be used to classify unseen instances? The answer is that because it has a bias towards smaller trees. This bias is a preference bias in which the algorithm prefers smaller trees.

If the data set had elements with many attributes then the hypotheses space would be large. One can then alter the termination criterion that prevents the tree from becoming too deep. This could be done by stopping the partitioning process as soon as the majority of the elements in a node are of one class. One can also include a pruning step that removes nodes with a very small number of training instances.

The algorithm ID3 is described in Figure 18.12. The algorithm is recursive in nature in which subtrees are constructed recursively. At each node, the attribute that partitions the associated set with maximum separation of the different classes is chosen. Observe that each attribute can be used only once. This is in contrast to *kd-trees*, where an attribute can be used more than once.

The *ID3* algorithm essentially does *Hill Climbing* (see Chapter 3) over the space of possible decision trees. It begins with the empty tree and

recursively builds subtrees in a greedy manner, choosing the most promising attribute for each node to partition the data. Like *Find-S* algorithm, and unlike *Candidate-Elimination*, it outputs a single hypothesis given a training set labelled with class data. Unlike both the algorithms, *ID3* is not sensitive to noise (errors) in the training data.

The algorithm *ID3* is designed to handle nominal attributes. In order to be able to use this algorithm, we manually converted the *Experience* attribute which had numeric data to *Exp-Symb*, which had three nominal values *Low*, *Medium* and *High*. The algorithm *C4.5*, and *C5*, devised also by Ross Quinlan (1992) extends *ID3* to handle numeric data as well. Algorithm *C4.5* improves upon *ID3* in other ways as well. It is more robust in handling noisy data, and can process data with missing values too. It produces rules directly and allows post pruning of the rules to avoid 'overfitting', by controlling how deep the tree grows.

The algorithm *C4.5* handles attributes with numeric (or continuous) data by placing split points at appropriate locations, usually close to the half-way mark. If one were to arrange the elements in increasing order of the attribute value then a split point is never placed between two values, corresponding to elements having the same class label. The algorithm in fact tries out different possible split points and chooses the one that maximizes information gain.

Figure 18.13 shows the decision tree constructed for our example, from the original data set in Table 15.3. The set of attributes *Att* used here is {*Age*, *Education*, *Experience*, *Hands-On*}, with *Age* being the new attribute added.

```
ID3 (Training Set: T, Target-attribute: C, Attributes: Att)
 1 create node N with set T
 2 if all instances of T in N have class label c∈values(C)
 3    then return N with label c
 4 if empty(Att)
 5    then return N with majority label from T
 6 A ← chooseAttribute(Att, T)     /* with maximum information gain */
 7 N.decisionAttribute ← A
 8 Att ← Att - {A}
 9 for each value V of A
10     T_V ← subset of T with value V of elements
11     create node N_V with set T_V
12     child(N,V) ← N_V           /* the child of N with value V */
13     if empty(T_V)
14          then
15              label N_V with majority label from T
16          else
17              N_V ← ID3(T_V, C, Att)
18 return N
```

FIGURE 18.12 The procedure *ID3* creates a tree structure with a decision point at each node. At every node, the attribute that maximizes information gain is chosen to partition the set based on the attribute values, and each partition is processed recursively. Function *chooseAttribute*(*Att*, *T*) chooses the maximum information gain attribute from the set *Att*, given the training set *T*.

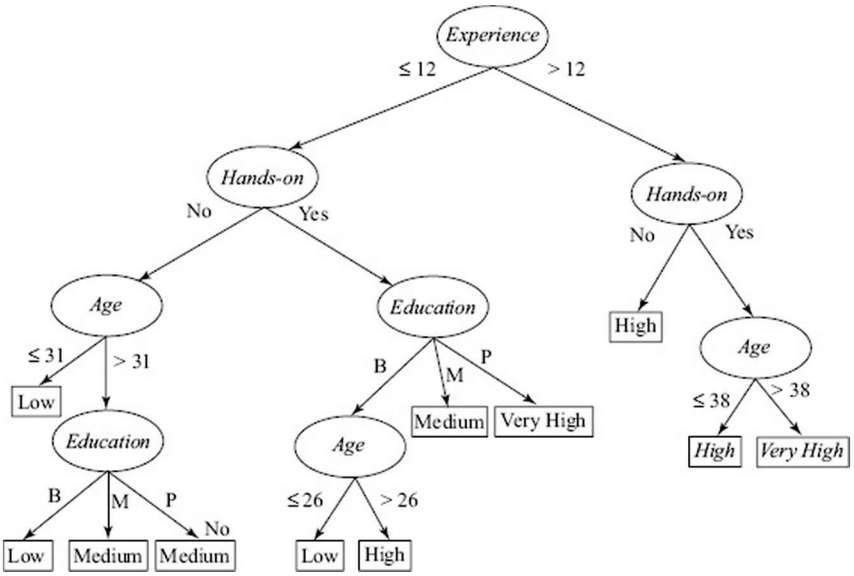

FIGURE 18.13 The decision tree constructed by C4.5 for the date set with numerical values in Table 15.3. An additional numeric attribute *Age* has also been considered.

It is interesting to note that with these numerical values too, *Experience* has been the attribute that is chosen first, though the algorithm is designed to make a two-way split only. Again, here the *Hands-On* attribute plays a significant role in classifying elements. The new attribute *Age* too makes an appearance in some places.

The reader is encouraged to add one more attribute *Gender* from Table 15.3 and investigate whether that plays a role too. Implementations of the algorithms *ID3*, *C4.5 and C5.0* are available on the Web[6], for those not inclined to try their own coding skills.

In supervised learning, a teacher is showing the true class labels to the learning algorithm. In the absence of a teacher too, learning is possible. We look at two approaches to do so. One is when the learner ponders over the samples and tries to cluster them into similar samples. The process is called clustering, and happens in an unsupervised manner. It could be a first step towards forming concepts which can be articulated. The other is when the learner learns from the outcome of its actions when acting in an environment. By observing the (sequences of) actions that lead to a favourable outcome or an unfavourable one, the learner can tune its decision making algorithm to be biased towards actions that are more rewarding.

18.6 The *K-means* Clustering Algorithm[7]

The focus of this chapter till now has been supervised learning. Now, we look at clustering, a classical unsupervised learning problem. Consider a

set of data points, $X = \{x_1, x_2, \ldots, x_n\}$ in a Euclidean space where the distance between any pair of objects(x_i, x_j) is computed as the *L2* norm and is denoted by $d(x_i,x_j)$. Clustering is the task of grouping the data points into groups, such that the data points in the same group are more "*similar*" than data points across groups. Such groups discovered by a clustering technique are referred to as *clusters*. The algorithm *K-means* for clustering partitions a data set into *k* clusters, where *k* is a user defined number. It was first devised by Stuart Lloyd (1957) and is also known as Llyod's algorithm. The clusters generated by *K-means* are disjoint, in that each data point would be uniquely assigned to one cluster; further, algorithm *K-means* ensures that no points are left unassigned.

In the case of a clustering of our dataset *X*, we may define the clustering problem declaratively, based on the desired properties of the groups output by the clustering as follows: *If one picks a random pair of objects that both belong to the same group* (*cluster*) *and another pair of objects that belong to different groups*, *it should be highly likely that the distance between the former pair is much lesser than the distance between the latter pair*.

K-means attempts to achieve such a property in generating *k* groups from the dataset $X = \{x_1, x_2, \ldots, x_n\}$, in an iterative fashion. After any iteration, there would be a set of *k* clusters, that we denote by $\{C_1, C_2, \ldots, C_k\}$. Within each iteration, *K-means* refines the clusters from the previous iteration by optimizing the following objective function (for all the clusters):

$$O(cluster(.), \mu(.)) = \sum_{1 \le i \le n} (d(x_i, \mu(cluster(x_i))))^2 \qquad (18.47)$$

where $cluster(x_i)$ denotes[8] the cluster to which x_i belongs and $\mu(c)$ denotes a representative point, or the *centroid*, for the cluster *c*, and *d*(.,.) denotes the distance function (*L2* norm). That is, the algorithm strives to minimize the sum of the distances of each point to the centroid of the cluster to which the point belongs. In order to minimize this objective function, we could change one or both of the following, moving from one iteration to the next:

1. Assignments of data points to clusters, i.e., cluster(.) function
2. Choosing the centroid for a cluster, i.e., μ(.) function.

K-means adopts the simple approach of modifying these in sequence within each iteration. In particular, the cluster assignments, *cluster*(.), is modified keeping the μ(.) fixed, following by modifying μ(.) keeping the cluster assignments fixed.

Modifying the Cluster Assignments

We will now look at how the cluster assignments may be modified when the μ(.) function, the representative point for each cluster, is specified. For every point x_i,, the contribution to the objective function is directly

related to the distance to the representative point for the corresponding cluster. The obvious way to minimize the contribution of x_i to the objective function is by re-assigning x_i to the cluster whose centroid is nearest to it.

Thus,

$$cluster(x_i) = \underset{j}{\operatorname{argmin}}\, d(x_i, \mu(C_j)) \tag{18.48}$$

Modifying the Representative Points

We now consider modifying the representative points for each cluster. The contribution of each cluster C_j, to the objective function is the sum of the distance of each data point in C_j to the representative point, $\mu(C_j)$. This sum is directly related to the average distance of points in C_j to $\mu(C_j)$, which is intuitively minimized by setting $\mu(C_j)$ itself to the average (mean) of points in C_j.

Thus, each attribute of $\mu(C_j)$ would be assigned to the mean value of the attribute among objects in C_j. In particular, for the i^{th} attribute,

$$\mu(C_j)[i] = \frac{\sum_{x \in C_j} x[i]}{|C_j|} \tag{18.49}$$

Having described the basic steps that are performed in *K-means* clustering, we now outline the algorithm in Figure 18.14. It starts off with initializing $\mu(.)$ randomly for each cluster (Line 1), followed by a sequence of iterations in which the *cluster*(.) and $\mu(.)$ values are modified in sequence (lines 2–4).

```
K-Means(Set of Objects X = {x1, x2, …, xn}, #clusters k)
   1 Initialize μ(C1) through μ(Ck) randomly in the Euclidean space of X
   2 while O(cluster(.),μ(.)) has not yet converged
   3       do ∀i, 1 ≤ i ≤ n, cluster(xi) = argmin_j d(xi, μ(Cj))
   4       do ∀j, 1 ≤ j ≤ k, μ(Cj) = mean{x|cluster(x) = Cj}
   5 return cluster(.)
```

FIGURE 18.14 *K-Means* Algorithm iteratively assigns objects to clusters, and re-computes the cluster centroids till the objective function converges.

K-Means is one of the classical examples of the Expectation-Maximization technique briefly described in Section 18.3.1. Here, the *E* step corresponds to assigning data points to clusters (Line 3 in Figure 18.14), whereas the *M* step (in Line 4) corresponds to obtaining a new model for the clusters, i.e., a new set of $\mu(.)$ values. As we have seen, each of these steps are guaranteed to improve (minimize) the objective function $O(cluster(.),\mu(.))$ and the algorithm is set to converge when the improvements cease or are no longer significant.

We look an example where the algorithm is used.

Example: *Gene Clustering*

Genes store codes for making proteins, which carry out intended functions inside a cell. The protein synthesis from the gene takes place in two steps: (i) *transcription step*, in which messenger RNA (m-RNA) is formed by reading the gene of the DNA, and (ii) *translation step* that synthesizes the protein using amino acids mentioned in the gene sequence. Once a gene is transcribed into m-RNA, we say that the gene is expressed in the cell. The genes are expressed under specific physiological condition, in which particular proteins are required to carry out the specific task. The gene expression is measured using microarray technology. The dataset used in this illustration is constructed from two microarray experiments involving 10 genes, as shown in Table 18.5. Each gene has two features, $x^{(1)}$ and $x^{(2)}$, that correspond to expression levels of a gene under two different conditions, as measured by two microarray experiments.

The challenge in *K-means* clustering is to specify a number of clusters K as an input parameter to the algorithm. Visualization of the data provides useful information about distribution of points (genes in this case) in the input space (here, expression levels). Visualization is possible only when the number of dimensions is less than three. In this case, the number of dimensions is two and hence we can visualize the data. The visualization (Figure 18.15) shows that there are two natural groups in the data and hence we set $k = 2$ for this dataset.

Table 18.5 A small gene data set

Gene	*Exp. level* $x^{(1)}$	*Exp level* $x^{(2)}$
gene-1	1	1
gene-2	1	2
gene-3	2	2
gene-4	2	1
gene-5	3	2
gene-6	4	4
gene-7	4	5
gene-8	5	5
gene-9	5	6
gene-10	6	4

Whenever the number of dimensions is greater than three, it is usual practice to reduce the number of dimensions by projecting the data on lower dimensional subspace using techniques like Principle Component Analysis (PCA) or Multi-Dimensional Scaling (MDS). The visualization is performed using top two or three dimensions. We will not cover dimensionality reduction in this chapter and we refer interested readers to (Bishop, 2006).

The *K-means* algorithm starts by randomly initializing cluster centroids. Let μ_1 and μ_2 be the centroids of clusters 1 and 2 respectively.

Let's initialize two centroids to the following values:

$\mu_1 = (2, 2)$ and $\mu_2 = (5, 5)$

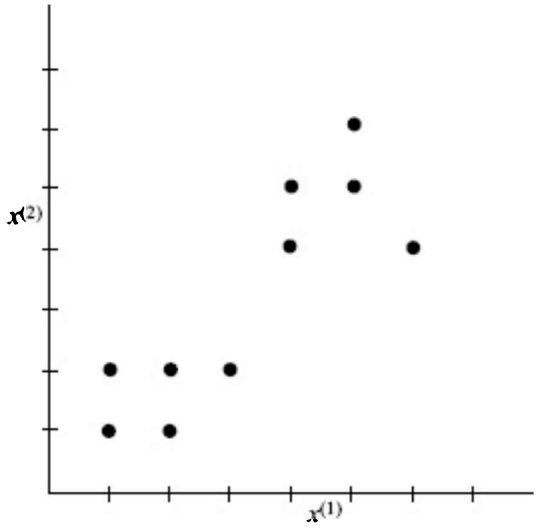

FIGURE 18.15 The gene expression data from Table 18.5.

It then follows an iterative process which runs till convergence, which is evaluated based on one of the two criteria:

- Cluster assignments do not change across successive iterations.
- Cluster centroids do not change across successive iterations.

Note that both these criteria are equivalent.

Iteration 1

E-step: Calculate the distance between each point to the cluster centroids, using Euclidean distance formula. The distances are shown in Table 18.6.

Table 18.6 The Euclidean distance of each point to the two centroids

Gene	$\mu_1 = (2,2)$	$\mu_2 = (5,5)$
gene-1	√2	√32
gene-2	1	5
gene-3	0	√18
gene-4	1	5
gene-5	1	√13
gene-6	√8	√2
gene-7	√13	1
gene-8	√18	0
gene-9	5	1
gene-10	√20	√

Table 18.7 The cluster assignment based on the distance in Table 18.6

Gene	*Cluster(gene-i)*
gene-1	1
gene-2	1
gene-3	1
gene-4	1
gene-5	1
gene-6	2
gene-7	2
gene-8	2
gene-9	2
gene-10	2

Based on the above distances, the ten genes are assigned to the closest cluster as shown in Table 18.7.

M-step: Re-estimate centroids of the two clusters. The genes 1 to 5 are assigned to cluster 1, while genes 6 to 10 are assigned to cluster 2. Based on this assignment, we re-estimate cluster centroids. The new centroids are as follows:

$\mu_1 = (1.8, 1.6)$ and $\mu_2 = (4.8, 4.8)$

Next, we check the convergence criteria as mentioned before in Line 2 of the algorithm. Since there is a change in cluster assignment, we conclude the iterative procedure is not converged and hence we continue with the iterations.

Iteration 2

E-step: Calculate the distance between each point to the cluster

centroids, using the Euclidean distance formula (Table 18.8).

It can be seen that the assignments of genes to clusters remains the same here.

M-step: Re-estimate centroids of each of the *K* clusters. When we recomputed the centroids of the two clusters, we found that the centroids do not change as well.

$$\mu_1 = (1.8, 1.6) \text{ and } \mu_2 = (4.8, 4.8)$$

Since the cluster labels of each object (gene) remains unchanged, as does the centroid for each cluster, the iterative procedure has converged. Genes 1 to 5 form one cluster with centroid $\mu_1 = (1.8, 1.6)$ and genes 6–10 for the second cluster with centroid $\mu_2 = (4.8, 4.8)$.

Table 18.8 The Euclidean distance of each point to the two centroids, after iteration 2

Gene	$\mu_1 = (1.8,1.6)$	$\mu_2 = (4.8,4.8)$
gene-1	1.0	5.4
gene-2	0.9	4.7
gene-3	0.4	4.0
gene-4	0.6	4.7
gene-5	1.3	3.3
gene-6	3.3	1.1
gene-7	4.0	0.8
gene-8	4.7	0.3
gene-9	5.4	1.2
gene-10	4.8	1.4

18.7 Learning from Outcomes

We began this chapter with supervised learning, in which a learning system works with labelled data in a supervised learning process. The learning system then learns to assign the label for similar unseen data. We then saw an unsupervised, learning algorithm which observes a given data set and partitions it into subsets of similar elements, based on some measure of distance or similarity.[9]

An autonomous agent operating a world has other opportunities to learn as well. The agent can observe what happens after it carries out an action or a set of actions in the world, and learn to identify those actions that are more rewarding. For actions that yield an immediate reward, the learning task is relatively straightforward, as will be corroborated/vouched by anyone who plonks down in front of a television set to watch a football game, where finishing the class assignment would have yielded a reward somewhat in the future. We had briefly touched upon the problem of such goal conflicts in Section 14.5. However, this is not the topic of our focus here. Instead, we look briefly at how an agent can learn to identify actions when the reward from them is given sometime in the future, and in the