

# Types of Machine Learning Algorithms

Taiwo Oladipupo Ayodele  
*University of Portsmouth*  
*United Kingdom*

## 1. Machine Learning: Algorithms Types

Machine learning algorithms are organized into taxonomy, based on the desired outcome of the algorithm. Common algorithm types include:

- Supervised learning --- where the algorithm generates a function that maps inputs to desired outputs. One standard formulation of the supervised learning task is the classification problem: the learner is required to learn (to approximate the behavior of) a function which maps a vector into one of several classes by looking at several input-output examples of the function.
- Unsupervised learning --- which models a set of inputs: labeled examples are not available.
- Semi-supervised learning --- which combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- Reinforcement learning --- where the algorithm learns a policy of how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback that guides the learning algorithm.
- Transduction --- similar to supervised learning, but does not explicitly construct a function: instead, tries to predict new outputs based on training inputs, training outputs, and new inputs.
- Learning to learn --- where the algorithm learns its own inductive bias based on previous experience.

The performance and computational analysis of machine learning algorithms is a branch of statistics known as computational learning theory.

Machine learning is about designing algorithms that allow a computer to learn. Learning is not necessarily involves consciousness but learning is a matter of finding statistical regularities or other patterns in the data. Thus, many machine learning algorithms will barely resemble how human might approach a learning task. However, learning algorithms can give insight into the relative difficulty of learning in different environments.

### 1.1 Supervised Learning Approach

Supervised learning<sup>1</sup> is fairly common in classification problems because the goal is often to get the computer to learn a classification system that we have created. Digit recognition, once again, is a common example of classification learning. More generally, classification learning is appropriate for any problem where deducing a classification is useful and the classification is easy to determine. In some cases, it might not even be necessary to give pre-determined classifications to every instance of a problem if the agent can work out the classifications for itself. This would be an example of unsupervised learning in a classification context.

Supervised learning<sup>2</sup> often leaves the probability for inputs undefined. This model is not needed as long as the inputs are available, but if some of the input values are missing, it is not possible to infer anything about the outputs. Unsupervised learning, all the observations are assumed to be caused by latent variables, that is, the observations is assumed to be at the end of the causal chain. Examples of supervised learning and unsupervised learning are shown in the figure 1 below:

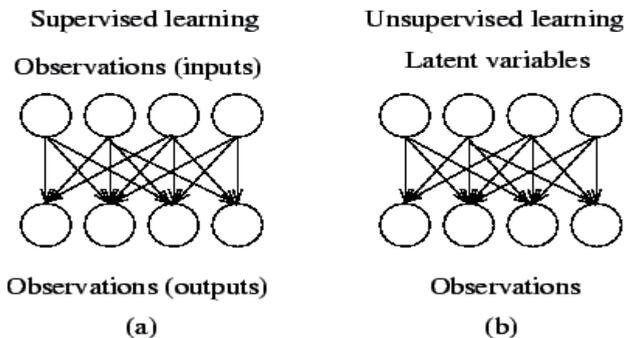


Fig. 1. Examples of Supervised and Unsupervised Learning

Supervised learning<sup>3</sup> is the most common technique for training neural networks and decision trees. Both of these techniques are highly dependent on the information given by the pre-determined classifications. In the case of neural networks, the classification is used to determine the error of the network and then adjust the network to minimize it, and in decision trees, the classifications are used to determine what attributes provide the most information that can be used to solve the classification puzzle. We'll look at both of these in more detail, but for now, it should be sufficient to know that both of these examples thrive on having some "supervision" in the form of pre-determined classifications.

Inductive machine learning is the process of learning a set of rules from instances (examples in a training set), or more generally speaking, creating a classifier that can be used to generalize from new instances. The process of applying supervised ML to a real-world problem is described in Figure F. The first step is collecting the dataset. If a requisite expert is available, then s/he could suggest which fields (attributes, features) are the most

<sup>1</sup> [http://www.aihorizon.com/essays/generalai/supervised\\_unsupervised\\_machine\\_learning.htm](http://www.aihorizon.com/essays/generalai/supervised_unsupervised_machine_learning.htm)

<sup>2</sup> [http://www.cis.hut.fi/harri/thesis/valpola\\_thesis/node34.html](http://www.cis.hut.fi/harri/thesis/valpola_thesis/node34.html)

<sup>3</sup> [http://www.aihorizon.com/essays/generalai/supervised\\_unsupervised\\_machine\\_learning.htm](http://www.aihorizon.com/essays/generalai/supervised_unsupervised_machine_learning.htm)

informative. If not, then the simplest method is that of "brute-force," which means measuring everything available in the hope that the right (informative, relevant) features can be isolated. However, a dataset collected by the "brute-force" method is not directly suitable for induction. It contains in most cases noise and missing feature values, and therefore requires significant pre-processing according to Zhang et al (Zhang, 2002).

The second step is the data preparation and data pre-processing. Depending on the circumstances, researchers have a number of methods to choose from to handle missing data (Batista, 2003). Hodge et al (Hodge, 2004) , have recently introduced a survey of contemporary techniques for outlier (noise) detection. These researchers have identified the techniques' advantages and disadvantages. Instance selection is not only used to handle noise but to cope with the infeasibility of learning from very large datasets. Instance selection in these datasets is an optimization problem that attempts to maintain the mining quality while minimizing the sample size. It reduces data and enables a data mining algorithm to function and work effectively with very large datasets. There is a variety of procedures for sampling instances from a large dataset. See figure 2 below.

Feature subset selection is the process of identifying and removing as many irrelevant and redundant features as possible (Yu, 2004) . This reduces the dimensionality of the data and enables data mining algorithms to operate faster and more effectively. The fact that many features depend on one another often unduly influences the accuracy of supervised ML classification models. This problem can be addressed by constructing new features from the basic feature set. This technique is called feature construction/transformation. These newly generated features may lead to the creation of more concise and accurate classifiers. In addition, the discovery of meaningful features contributes to better comprehensibility of the produced classifier, and a better understanding of the learned concept. Speech recognition using hidden Markov models and Bayesian networks relies on some elements of supervision as well in order to adjust parameters to, as usual, minimize the error on the given inputs. Notice something important here: in the classification problem, the goal of the learning algorithm is to minimize the error with respect to the given inputs. These inputs, often called the "training set", are the examples from which the agent tries to learn. But learning the training set well is not necessarily the best thing to do. For instance, if I tried to teach you exclusive-or, but only showed you combinations consisting of one true and one false, but never both false or both true, you might learn the rule that the answer is always true. Similarly, with machine learning algorithms, a common problem is over-fitting the data and essentially memorizing the training set rather than learning a more general classification technique. As you might imagine, not all training sets have the inputs classified correctly. This can lead to problems if the algorithm used is powerful enough to memorize even the apparently "special cases" that don't fit the more general principles. This, too, can lead to over fitting, and it is a challenge to find algorithms that are both powerful enough to learn complex functions and robust enough to produce generalisable results.

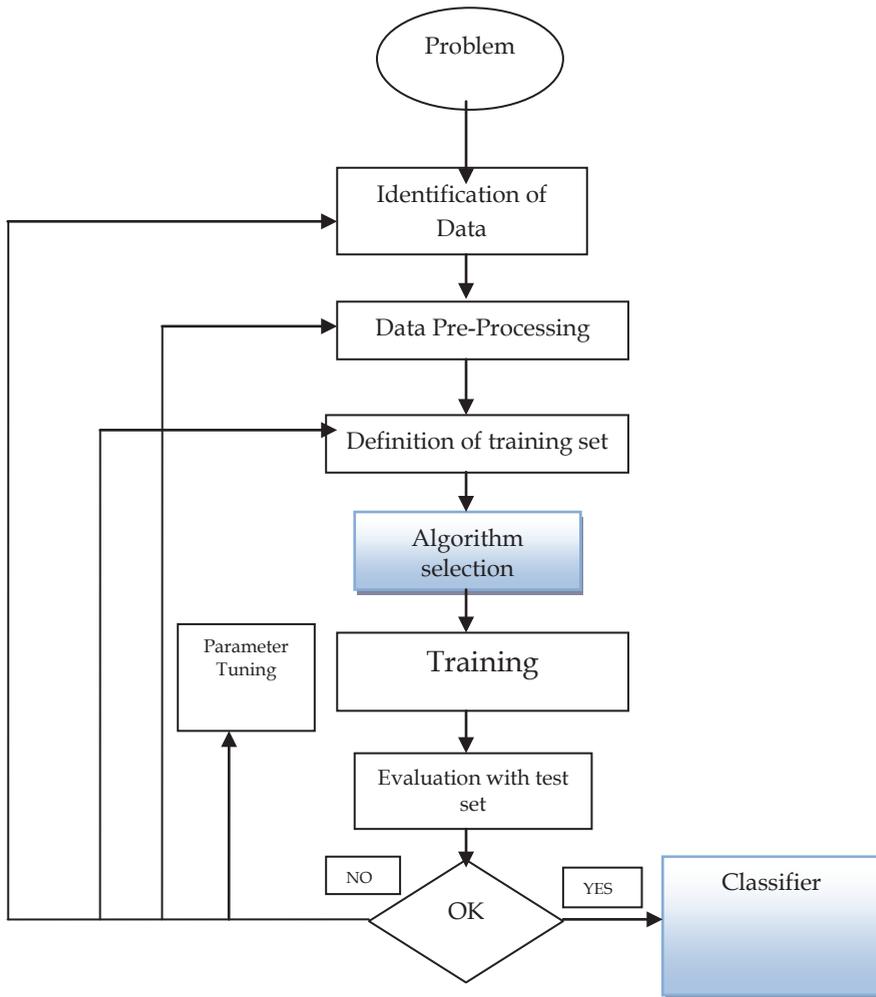


Fig. 2. Machine Learning Supervise Process

### 1.2 Unsupervised learning

Unsupervised learning<sup>4</sup> seems much harder: the goal is to have the computer learn how to do something that we don't tell it how to do! There are actually two approaches to unsupervised learning. The first approach is to teach the agent not by giving explicit categorizations, but by using some sort of reward system to indicate success. Note that this type of training will generally fit into the decision problem framework because the goal is not to produce a classification but to make decisions that maximize rewards. This approach nicely generalizes to the real world, where agents might be rewarded for doing certain

<sup>4</sup> [http://www.aihorizon.com/essays/generalai/supervised\\_unsupervised\\_machine\\_learning.htm](http://www.aihorizon.com/essays/generalai/supervised_unsupervised_machine_learning.htm)

actions and punished for doing others. Often, a form of reinforcement learning can be used for unsupervised learning, where the agent bases its actions on the previous rewards and punishments without necessarily even learning any information about the exact ways that its actions affect the world. In a way, all of this information is unnecessary because by learning a reward function, the agent simply knows what to do without any processing because it knows the exact reward it expects to achieve for each action it could take. This can be extremely beneficial in cases where calculating every possibility is very time consuming (even if all of the transition probabilities between world states were known). On the other hand, it can be very time consuming to learn by, essentially, trial and error. But this kind of learning can be powerful because it assumes no pre-discovered classification of examples. In some cases, for example, our classifications may not be the best possible. One striking example is that the conventional wisdom about the game of backgammon was turned on its head when a series of computer programs (neuro-gammon and TD-gammon) that learned through unsupervised learning became stronger than the best human chess players merely by playing themselves over and over. These programs discovered some principles that surprised the backgammon experts and performed better than backgammon programs trained on pre-classified examples. A second type of unsupervised learning is called clustering. In this type of learning, the goal is not to maximize a utility function, but simply to find similarities in the training data. The assumption is often that the clusters discovered will match reasonably well with an intuitive classification. For instance, clustering individuals based on demographics might result in a clustering of the wealthy in one group and the poor in another. Although the algorithm won't have names to assign to these clusters, it can produce them and then use those clusters to assign new examples into one or the other of the clusters. This is a data-driven approach that can work well when there is sufficient data; for instance, social information filtering algorithms, such as those that Amazon.com use to recommend books, are based on the principle of finding similar groups of people and then assigning new users to groups. In some cases, such as with social information filtering, the information about other members of a cluster (such as what books they read) can be sufficient for the algorithm to produce meaningful results. In other cases, it may be the case that the clusters are merely a useful tool for a human analyst. Unfortunately, even unsupervised learning suffers from the problem of overfitting the training data. There's no silver bullet to avoiding the problem because any algorithm that can learn from its inputs needs to be quite powerful.

Unsupervised learning algorithms according to Ghahramani (Ghahramani, 2008) are designed to extract structure from data samples. The quality of a structure is measured by a cost function which is usually minimized to infer optimal parameters characterizing the hidden structure in the data. Reliable and robust inference requires a guarantee that extracted structures are typical for the data source, i.e., similar structures have to be extracted from a second sample set of the same data source. Lack of robustness is known as over fitting from the statistics and the machine learning literature. In this talk I characterize the over fitting phenomenon for a class of histogram clustering models which play a prominent role in information retrieval, linguistic and computer vision applications. Learning algorithms with robustness to sample fluctuations are derived from large deviation results and the maximum entropy principle for the learning process.

Unsupervised learning has produced many successes, such as world-champion calibre backgammon programs and even machines capable of driving cars! It can be a powerful technique when there is an easy way to assign values to actions. Clustering can be useful when there is enough data to form clusters (though this turns out to be difficult at times) and especially when additional data about members of a cluster can be used to produce further results due to dependencies in the data. Classification learning is powerful when the classifications are known to be correct (for instance, when dealing with diseases, it's generally straight-forward to determine the design after the fact by an autopsy), or when the classifications are simply arbitrary things that we would like the computer to be able to recognize for us. Classification learning is often necessary when the decisions made by the algorithm will be required as input somewhere else. Otherwise, it wouldn't be easy for whoever requires that input to figure out what it means. Both techniques can be valuable and which one you choose should depend on the circumstances--what kind of problem is being solved, how much time is allotted to solving it (supervised learning or clustering is often faster than reinforcement learning techniques), and whether supervised learning is even possible.

### 1.3 Algorithm Types

In the area of supervised learning which deals much with classification. These are the algorithms types:

- Linear Classifiers
  - Logical Regression
  - Naïve Bayes Classifier
  - Perceptron
  - Support Vector Machine
- Quadratic Classifiers
- K-Means Clustering
- Boosting
- Decision Tree
  - Random Forest
- Neural networks
- Bayesian Networks

**Linear Classifiers:** In machine learning, the goal of classification is to group items that have similar feature values, into groups. Timothy et al (Timothy Jason Shepard, 1998) stated that a linear classifier achieves this by making a classification decision based on the value of the linear combination of the features. If the input feature vector to the classifier is a real vector  $\vec{x}$ , then the output score is

$$y = f(\vec{w} \cdot \vec{x}) = f\left(\sum_j w_j x_j\right),$$

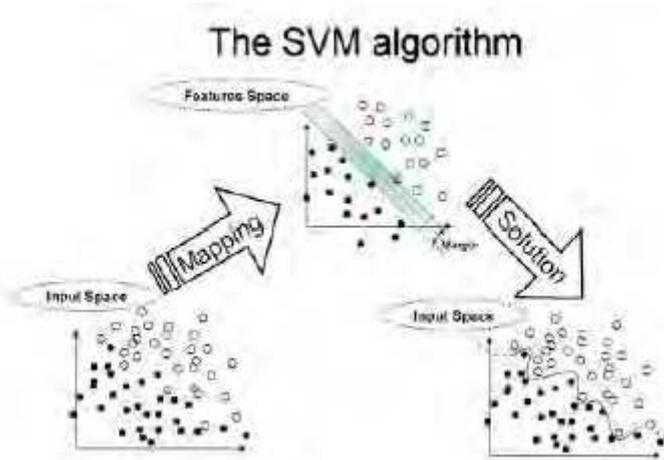
where  $\vec{w}$  is a real vector of weights and  $f$  is a function that converts the dot product of the two vectors into the desired output. The weight vector  $\vec{w}$  is learned from a set of labelled training samples. Often  $f$  is a simple function that maps all values above a certain threshold to the first class and all other values to the second class. A more complex  $f$  might give the probability that an item belongs to a certain class.

For a two-class classification problem, one can visualize the operation of a linear classifier as splitting a high-dimensional input space with a hyperplane: all points on one side of the hyper plane are classified as "yes", while the others are classified as "no". A linear classifier is often used in situations where the speed of classification is an issue, since it is often the fastest classifier, especially when  $\vec{x}$  is sparse. However, decision trees can be faster. Also, linear classifiers often work very well when the number of dimensions in  $\vec{x}$  is large, as in document classification, where each element in  $\vec{x}$  is typically the number of counts of a word in a document (see document-term matrix). In such cases, the classifier should be well-regularized.

- **Support Vector Machine:** A Support Vector Machine as stated by Luis et al (Luis Gonz, 2005) (SVM) performs classification by constructing an  $N$ -dimensional hyper plane that optimally separates the data into two categories. SVM models are closely related to neural networks. In fact, a SVM model using a sigmoid kernel function is equivalent to a two-layer, perceptron neural network.

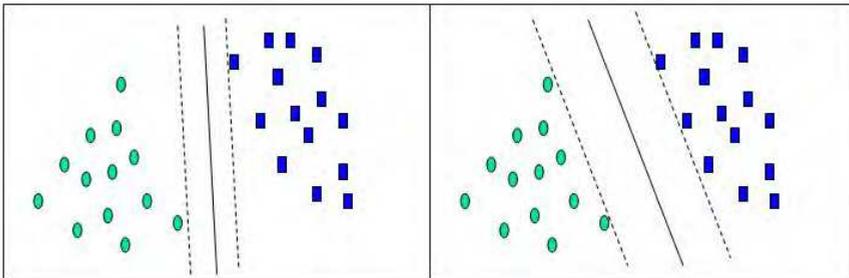
Support Vector Machine (SVM) models are a close cousin to classical multilayer perceptron neural networks. Using a kernel function, SVM's are an alternative training method for polynomial, radial basis function and multi-layer perceptron classifiers in which the weights of the network are found by solving a quadratic programming problem with linear constraints, rather than by solving a non-convex, unconstrained minimization problem as in standard neural network training.

In the parlance of SVM literature, a predictor variable is called an *attribute*, and a transformed attribute that is used to define the hyper plane is called a *feature*. The task of choosing the most suitable representation is known as *feature selection*. A set of features that describes one case (i.e., a row of predictor values) is called a *vector*. So the goal of SVM modelling is to find the optimal hyper plane that separates clusters of vector in such a way that cases with one category of the target variable are on one side of the plane and cases with the other category are on the other size of the plane. The vectors near the hyper plane are the *support vectors*. The figure below presents an overview of the SVM process.



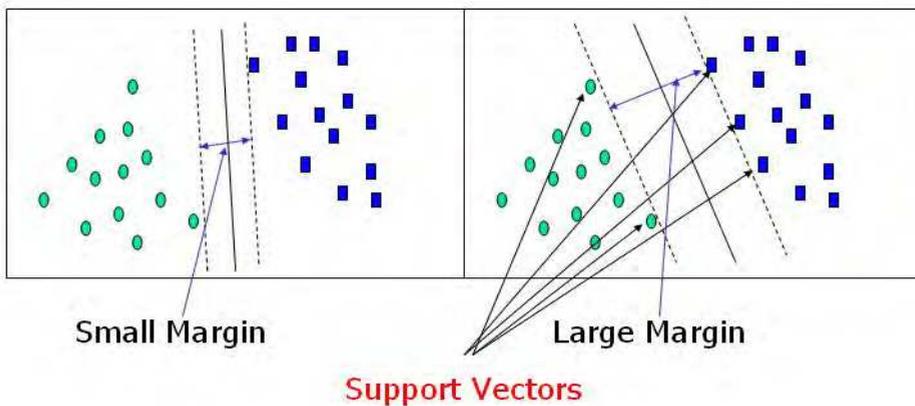
### A Two-Dimensional Example

Before considering  $N$ -dimensional hyper planes, let's look at a simple 2-dimensional example. Assume we wish to perform a classification, and our data has a categorical target variable with two categories. Also assume that there are two predictor variables with continuous values. If we plot the data points using the value of one predictor on the X axis and the other on the Y axis we might end up with an image such as shown below. One category of the target variable is represented by rectangles while the other category is represented by ovals.



In this idealized example, the cases with one category are in the lower left corner and the cases with the other category are in the upper right corner; the cases are completely separated. The SVM analysis attempts to find a 1-dimensional hyper plane (i.e. a line) that separates the cases based on their target categories. There are an infinite number of possible lines; two candidate lines are shown above. The question is which line is better, and how do we define the optimal line.

The dashed lines drawn parallel to the separating line mark the distance between the dividing line and the closest vectors to the line. The distance between the dashed lines is called the *margin*. The vectors (points) that constrain the width of the margin are the *support vectors*. The following figure illustrates this.



An SVM analysis (Luis Gonz, 2005) finds the line (or, in general, hyper plane) that is oriented so that the margin between the support vectors is maximized. In the figure above, the line in the right panel is superior to the line in the left panel.

If all analyses consisted of two-category target variables with two predictor variables, and the cluster of points could be divided by a straight line, life would be easy. Unfortunately, this is not generally the case, so SVM must deal with (a) more than two predictor variables, (b) separating the points with non-linear curves, (c) handling the cases where clusters cannot be completely separated, and (d) handling classifications with more than two categories.

In this chapter, we shall explain three main machine learning techniques with their examples and how they perform in reality. These are:

- K-Means Clustering
- Neural Network
- Self Organised Map

### 1.3.1 K-Means Clustering

The basic step of k-means clustering is uncomplicated. In the beginning we determine number of cluster  $K$  and we assume the centre of these clusters. We can take any random objects as the initial centre or the first  $K$  objects in sequence can also serve as the initial centre. Then the  $K$  means algorithm will do the three steps below until convergence.

Iterate until *stable* (= no object move group):

1. Determine the centre coordinate
2. Determine the distance of each object to the centre
3. Group the object based on minimum distance

The Figure 3 shows a K- means flow diagram

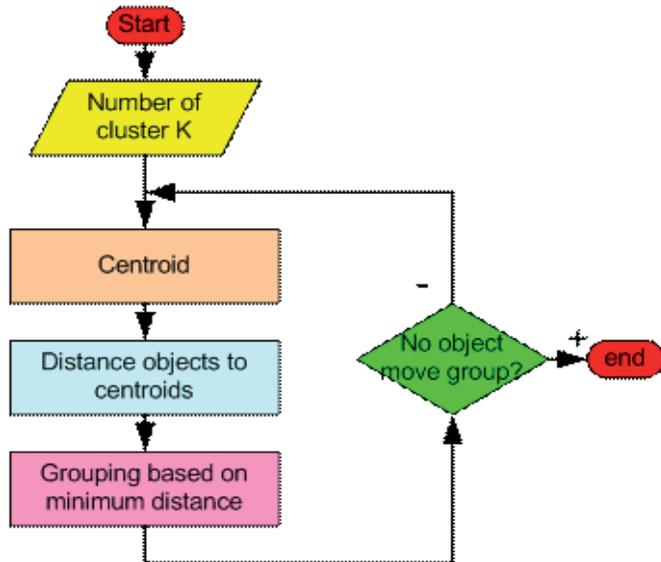


Fig. 3. K-means iteration

K-means (Bishop C. M., 1995) and (Tapas Kanungo, 2002) is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume  $k$  clusters) fixed a priori. The main idea is to define  $k$  centroids, one for each cluster. These centroids should be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate  $k$  new centroids as barycenters of the clusters resulting from the previous step. After we have these  $k$  new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the  $k$  centroids change their location step by step until no more changes are done. In other words centroids do not move any more.

Finally, this algorithm aims at minimizing an *objective function*, in this case a squared error function. The objective function

$$J = \sum_{j=1}^k \sum_{i=1}^n \|x_i^{(j)} - c_j\|^2$$

where  $\|x_i^{(j)} - c_j\|^2$  is a chosen distance measure between a data point  $x_i^{(j)}$  and the cluster centre  $c_j$ , is an indicator of the distance of the  $n$  data points from their respective cluster centres.

The algorithm in figure 4 is composed of the following steps:

1. Place  $K$  points into the space represented by the objects that are being clustered. These points represent initial group centroids.
2. Assign each object to the group that has the closest centroid.
3. When all objects have been assigned, recalculate the positions of the  $K$  centroids.
4. Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centres. The k-means algorithm can be run multiple times to reduce this effect. K-means is a simple algorithm that has been adapted to many problem domains. As we are going to see, it is a good candidate for extension to work with fuzzy feature vectors.

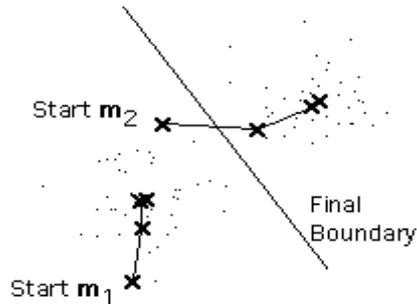
*An example*

Suppose that we have  $n$  sample feature vectors  $x_1, x_2, \dots, x_n$  all from the same class, and we know that they fall into  $k$  compact clusters,  $k < n$ . Let  $m_i$  be the mean of the vectors in cluster  $i$ . If the clusters are well separated, we can use a minimum-distance classifier to separate them. That is, we can say that  $x$  is in cluster  $i$  if  $\|x - m_i\|$  is the minimum of all the  $k$  distances. This suggests the following procedure for finding the  $k$  means:

- Make initial guesses for the means  $m_1, m_2, \dots, m_k$
- Until there are no changes in any mean
- Use the estimated means to classify the samples into clusters
- For  $i$  from 1 to  $k$

- Replace  $\mathbf{m}_i$  with the mean of all of the samples for cluster  $i$
- end\_for
- end\_until

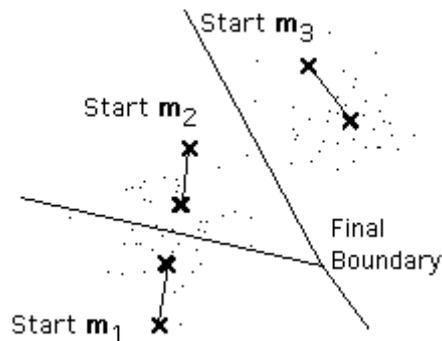
Here is an example showing how the means  $\mathbf{m}_1$  and  $\mathbf{m}_2$  move into the centers of two clusters.



This is a simple version of the k-means procedure. It can be viewed as a greedy algorithm for partitioning the  $n$  samples into  $k$  clusters so as to minimize the sum of the squared distances to the cluster centers. It does have some weaknesses:

- The way to initialize the means was not specified. One popular way to start is to randomly choose  $k$  of the samples.
- The results produced depend on the initial values for the means, and it frequently happens that suboptimal partitions are found. The standard solution is to try a number of different starting points.
- It can happen that the set of samples closest to  $\mathbf{m}_i$  is empty, so that  $\mathbf{m}_i$  cannot be updated. This is an annoyance that must be handled in an implementation, but that we shall ignore.
- The results depend on the metric used to measure  $||\mathbf{x} - \mathbf{m}_i||$ . A popular solution is to normalize each variable by its standard deviation, though this is not always desirable.
- The results depend on the value of  $k$ .

This last problem is particularly troublesome, since we often have no way of knowing how many clusters exist. In the example shown above, the same algorithm applied to the same data produces the following 3-means clustering. Is it better or worse than the 2-means clustering?



Unfortunately there is no general theoretical solution to find the optimal number of clusters for any given data set. A simple approach is to compare the results of multiple runs with different  $k$  classes and choose the best one according to a given criterion

### 1.3.2 Neural Network

Neural networks (Bishop C. M., 1995) can actually perform a number of regression and/or classification tasks at once, although commonly each network performs only one. In the vast majority of cases, therefore, the network will have a single output variable, although in the case of many-state classification problems, this may correspond to a number of output units (the post-processing stage takes care of the mapping from output units to output variables). If you do define a single network with multiple output variables, it may suffer from cross-talk (the hidden neurons experience difficulty learning, as they are attempting to model at least two functions at once). The best solution is usually to train separate networks for each output, then to combine them into an ensemble so that they can be run as a unit. Neural methods are:

- **Multilayer Perceptrons:** This is perhaps the most popular network architecture in use today, due originally to Rumelhart and McClelland (1986) and discussed at length in most neural network textbooks (e.g., Bishop, 1995). This is the type of network discussed briefly in previous sections: the units each perform a biased weighted sum of their inputs and pass this activation level through a transfer function to produce their output, and the units are arranged in a layered feed forward topology. The network thus has a simple interpretation as a form of input-output model, with the weights and thresholds (biases) the free parameters of the model. Such networks can model functions of almost arbitrary complexity, with the number of layers, and the number of units in each layer, determining the function complexity. Important issues in Multilayer Perceptrons (MLP) design include specification of the number of hidden layers and the number of units in these layers (Bishop C. M., 1995), (D. Michie, 1994).

The number of input and output units is defined by the problem (there may be some uncertainty about precisely which inputs to use, a point to which we will return later. However, for the moment we will assume that the input variables are

intuitively selected and are all meaningful). The number of hidden units to use is far from clear. As good a starting point as any is to use one hidden layer, with the number of units equal to half the sum of the number of input and output units. Again, we will discuss how to choose a sensible number later.

- **Training Multilayer Perceptrons:** Once the number of layers, and number of units in each layer, has been selected, the network's weights and thresholds must be set so as to minimize the prediction error made by the network. This is the role of the *training algorithms*. The historical cases that you have gathered are used to automatically adjust the weights and thresholds in order to minimize this error. This process is equivalent to fitting the model represented by the network to the training data available. The error of a particular configuration of the network can be determined by running all the training cases through the network, comparing the actual output generated with the desired or target outputs. The differences are combined together by an *error function* to give the network error. The most common error functions are the sum squared error (used for regression problems), where the individual errors of output units on each case are squared and summed together, and the cross entropy functions (used for maximum likelihood classification).

In traditional modeling approaches (e.g., linear modeling) it is possible to algorithmically determine the model configuration that absolutely minimizes this error. The price paid for the greater (non-linear) modeling power of neural networks is that although we can adjust a network to lower its error, we can never be sure that the error could not be lower still.

A helpful concept here is the error surface. Each of the  $N$  weights and thresholds of the network (i.e., the free parameters of the model) is taken to be a dimension in space. The  $N+1$ th dimension is the network error. For any possible configuration of weights the error can be plotted in the  $N+1$ th dimension, forming an *error surface*. The objective of network training is to find the lowest point in this many-dimensional surface.

In a linear model with sum squared error function, this error surface is a parabola (a quadratic), which means that it is a smooth bowl-shape with a single minimum. It is therefore "easy" to locate the minimum.

Neural network error surfaces are much more complex, and are characterized by a number of unhelpful features, such as local minima (which are lower than the surrounding terrain, but above the global minimum), flat-spots and plateaus, saddle-points, and long narrow ravines.

It is not possible to analytically determine where the global minimum of the error surface is, and so neural network training is essentially an exploration of the error surface. From an initially random configuration of weights and thresholds (i.e., a random point on the error surface), the training algorithms incrementally seek for the global minimum. Typically, the gradient (slope) of the error surface is calculated at the current point, and used to make a downhill move. Eventually, the algorithm stops in a low point, which may be a local minimum (but hopefully is the global minimum).

- **The Back Propagation Algorithm:** The best-known example of a neural network training algorithm is back propagation (Haykin, 19994), (Patterson, 19996), (Fausett, 19994). Modern second-order algorithms such as conjugate gradient descent and Levenberg-Marquardt (see Bishop, 1995; Shepherd, 1997) (both included in *ST Neural Networks*) are substantially faster (e.g., an order of magnitude faster) for many problems, but *back propagation* still has advantages in some circumstances, and is the easiest algorithm to understand. We will introduce this now, and discuss the more advanced algorithms later. In *back propagation*, the gradient vector of the error surface is calculated. This vector points along the line of steepest descent from the current point, so we know that if we move along it a "short" distance, we will decrease the error. A sequence of such moves (slowing as we near the bottom) will eventually find a minimum of some sort. The difficult part is to decide how large the steps should be. Large steps may converge more quickly, but may also overstep the solution or (if the error surface is very eccentric) go off in the wrong direction. A classic example of this in neural network training is where the algorithm progresses very slowly along a steep, narrow, valley, bouncing from one side across to the other. In contrast, very small steps may go in the correct direction, but they also require a large number of iterations. In practice, the step size is proportional to the slope (so that the algorithm settles down in a minimum) and to a special constant: the learning rate. The correct setting for the learning rate is application-dependent, and is typically chosen by experiment; it may also be time-varying, getting smaller as the algorithm progresses.

The algorithm is also usually modified by inclusion of a momentum term: this encourages movement in a fixed direction, so that if several steps are taken in the same direction, the algorithm "picks up speed", which gives it the ability to (sometimes) escape local minimum, and also to move rapidly over flat spots and plateaus.

The algorithm therefore progresses iteratively, through a number of epochs. On each epoch, the training cases are each submitted in turn to the network, and target and actual outputs compared and the error calculated. This error, together with the error surface gradient, is used to adjust the weights, and then the process repeats. The initial network configuration is random, and training stops when a given number of epochs elapses, or when the error reaches an acceptable level, or when the error stops improving (you can select which of these stopping conditions to use).

- **Over-learning and Generalization:** One major problem with the approach outlined above is that it doesn't actually minimize the error that we are really interested in - which is the expected error the network will make when *new* cases are submitted to it. In other words, the most desirable property of a network is its ability to *generalize* to new cases. In reality, the network is trained to minimize the error on the training set, and short of having a perfect and infinitely large training set, this is not the same thing as minimizing the error on the real error surface - the error surface of the underlying and unknown model (Bishop C. M., 1995).

The most important manifestation of this distinction is the problem of over-learning, or over-fitting. It is easiest to demonstrate this concept using polynomial curve fitting rather than neural networks, but the concept is precisely the same.

A polynomial is an equation with terms containing only constants and powers of the variables. For example:

$$y=2x+3$$

$$y=3x^2+4x+1$$

Different polynomials have different shapes, with larger powers (and therefore larger numbers of terms) having steadily more eccentric shapes. Given a set of data, we may want to fit a polynomial curve (i.e., a model) to explain the data. The data is probably noisy, so we don't necessarily expect the best model to pass exactly through all the points. A low-order polynomial may not be sufficiently flexible to fit close to the points, whereas a high-order polynomial is actually too flexible, fitting the data exactly by adopting a highly eccentric shape that is actually unrelated to the underlying function. See figure 4 below.

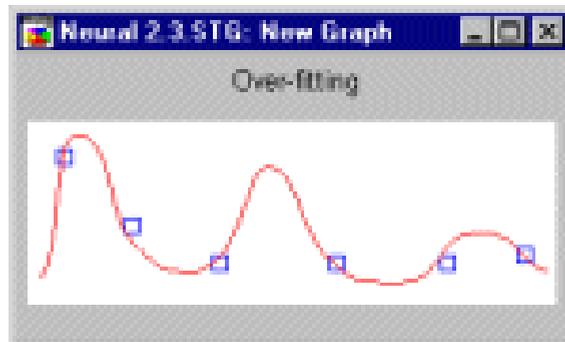


Fig. 4. High-order polynomial sample

Neural networks have precisely the same problem. A network with more weights models a more complex function, and is therefore prone to over-fitting. A network with less weight may not be sufficiently powerful to model the underlying function. For example, a network with no hidden layers actually models a simple linear function. How then can we select the right complexity of network? A larger network will almost invariably achieve a lower error eventually, but this may indicate over-fitting rather than good modeling.

The answer is to check progress against an independent data set, the selection set. Some of the cases are reserved, and not actually used for training in the back propagation algorithm. Instead, they are used to keep an independent check on the progress of the algorithm. It is invariably the case that the initial performance of the network on training and selection sets is the same (if it is not at least approximately the same, the division of cases between the two sets is probably biased). As training progresses, the training error naturally drops, and providing training is minimizing the true error function, the selection error drops too. However, if the selection error stops dropping, or indeed starts to rise, this indicates that the network is starting to overfit the data, and training should cease. When over-fitting occurs during the training process like this, it is called over-learning. In this case, it is usually

advisable to decrease the number of hidden units and/or hidden layers, as the network is over-powerful for the problem at hand. In contrast, if the network is not sufficiently powerful to model the underlying function, over-learning is not likely to occur, and neither training nor selection errors will drop to a satisfactory level.

The problems associated with local minima, and decisions over the size of network to use, imply that using a neural network typically involves experimenting with a large number of different networks, probably training each one a number of times (to avoid being fooled by local minima), and observing individual performances. The key guide to performance here is the selection error. However, following the standard scientific precept that, all else being equal, a simple model is always preferable to a complex model, you can also select a smaller network in preference to a larger one with a negligible improvement in selection error.

A problem with this approach of repeated experimentation is that the selection set plays a key role in selecting the model, which means that it is actually part of the training process. Its reliability as an independent guide to performance of the model is therefore compromised - with sufficient experiments, you may just hit upon a lucky network that happens to perform well on the selection set. To add confidence in the performance of the final model, it is therefore normal practice (at least where the volume of training data allows it) to reserve a third set of cases - the test set. The final model is tested with the test set data, to ensure that the results on the selection and training set are real, and not artifacts of the training process. Of course, to fulfill this role properly the test set should be used only once - if it is in turn used to adjust and reiterate the training process, it effectively becomes selection data!

This division into multiple subsets is very unfortunate, given that we usually have less data than we would ideally desire even for a single subset. We can get around this problem by resampling. Experiments can be conducted using different divisions of the available data into training, selection, and test sets. There are a number of approaches to this subset, including random (monte-carlo) resampling, cross-validation, and bootstrap. If we make design decisions, such as the best configuration of neural network to use, based upon a number of experiments with different subset examples, the results will be much more reliable. We can then either use those experiments solely to guide the decision as to which network types to use, and train such networks from scratch with new samples (this removes any sampling bias); or, we can retain the best networks found during the sampling process, but average their results in an ensemble, which at least mitigates the sampling bias.

To summarize, network design (once the input variables have been selected) follows a number of stages:

- Select an initial configuration (typically, one hidden layer with the number of hidden units set to half the sum of the number of input and output units).
- Iteratively conduct a number of experiments with each configuration, retaining the best network (in terms of selection error) found. A number of experiments are required with each configuration to avoid being fooled if training locates a local minimum, and it is also best to resample.
- On each experiment, if under-learning occurs (the network doesn't achieve an acceptable performance level) try adding more neurons to the hidden layer(s). If this doesn't help, try adding an extra hidden layer.

## Thank You for previewing this eBook

You can read the full version of this eBook in different formats:

- HTML (Free /Available to everyone)
- PDF / TXT (Available to V.I.P. members. Free Standard members can access up to 5 PDF/TXT eBooks per month each month)
- Epub & Mobipocket (Exclusive to V.I.P. members)

To download this full book, simply select the format you desire below

