A Tutorial on Machine Learning

By Sylvain Dindy-Bolongo

1.0 Introduction

Machine learning is concern with how to teach computers to learn from experience. Machine learning algorithms learn information directly from data. Such algorithms adaptively improve their performances as the number of available learning samples increases.
Machine learning algorithms find natural pattern in data that help make better decisions and predictions.

2.0 Machine Algorithms Overview

Machine learning algorithms work on data. The data is often preprocessed into features. Machine learning techniques consist in constructing some kind of model from collected data. Figure 2.1 shows the basic setup for statistical machine learning problem. The objective here is to model the true function f that transforms the input vector to some output. This function may be a regression problem or a category prediction problem.

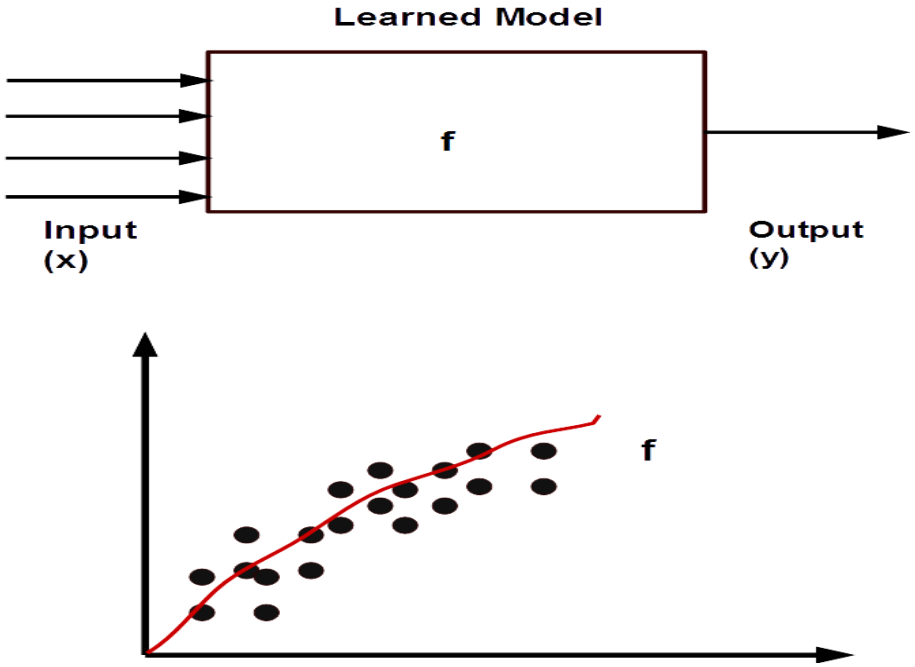


Figure 2.1 Setup for statistical machine learning: the classifier is trained to fit a data set. The model f is, most of the time corrupted by noise.

Machine learning algorithms usually have a set of goals to meet. In order to meet those goals, the learning algorithms analyzed collected features and adjust some parameters or weights. The process of parameter adjustment is what is called learning.

2.1 Training and Test Data Set

It is always important to know how well machine learning methods are working. In general the original data set is broken into large training set and smaller test set. The algorithm is run on the training set in order to learn the weight given the data feature vectors. After this is done, then test data are used in order to predict the outcome.

2.2 Supervised and Unsupervised Learning

The objective of supervised machine learning is to build a model to make prediction based on evidence in the presence of uncertainties. Supervised data has labels which are some kind of teaching signals that goes with the data feature vectors. If the data vectors are unlabeled the machine learning is unsupervised. The algorithm for such unsupervised learning are called clustering algorithms. In this situation, the goal is to groups unlabeled data vector that are closed.

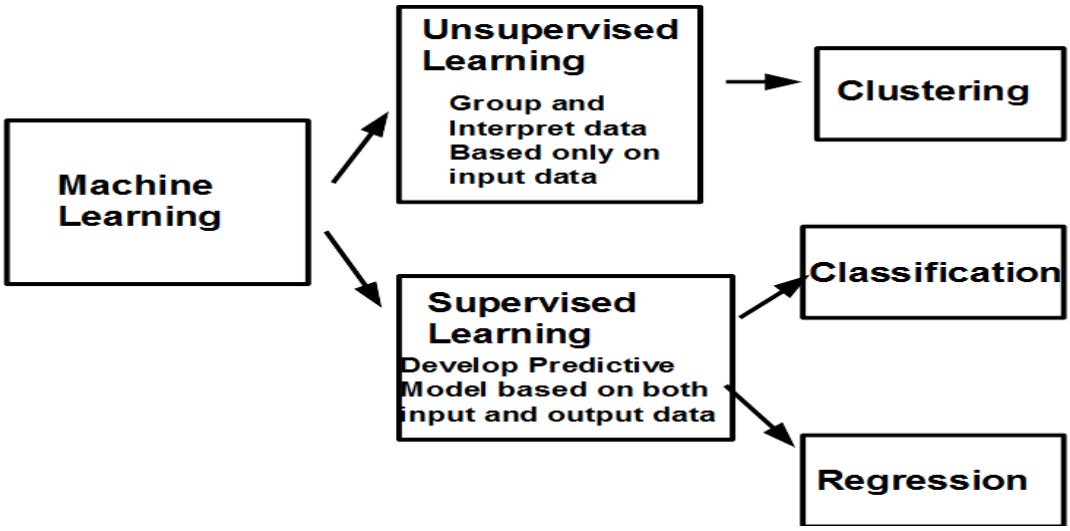


Figure 2.2 Supervised and unsupervised learning

Supervised learning can be categorical or the data can be numeric or ordered labels. When the data is numeric we say that we are doing regression. Supervised learning can involved one-to-one pairing of labels with data vector or it may consist of deferred learning (sometimes called reinforcement learning). In reinforcement learning, the data label (also called reward or punishment) can come long after the individual data vector is observed.

2.3 Discriminative and Generative Learning Algorithm

Algorithm that try to model the conditional probability $p(y \mid x)$ of y given x where y is the model output and x is the input feature is said to be discriminative learning algorithm and algorithm that try to model $p(y \mid x)$ and $p(y)$ a priori probability are called generative algorithms.

2.4 Feature dimension Reduction

We will examine two methods to reduce the input space dimension.

2.4.1 Principal component Analysis

The reduction is achieved by transforming the data into a new set of variable called principal components. In addition it is possible by means of eigen analysis to select only those principal components which preserve the most important feature of the original space.

The PCA methods is based on the linear transformation of N vector x into the vector y using the $N \times K$ matrix W and is given as follows:

$$y = Wx \tag{2.1}$$

Matrix W is defined as follows

$$W = [w_1, w_2, \ldots . w_k]^T \tag{2.2}$$

where the w_i are eigenvectors of the autocorrelation matrix of the input vector defined as follows:

$$\frac{1}{\dim(x)} xx^T \tag{2.3}$$

where $\dim(x)$ is the dimension of x.

The PCA transform, changes the large number of input data into a set of components according to their importance. The principal components are the projection of the original input vector x onto the principal directions of the eigenvectors. For example the 2D case is given on Figure 2.3.

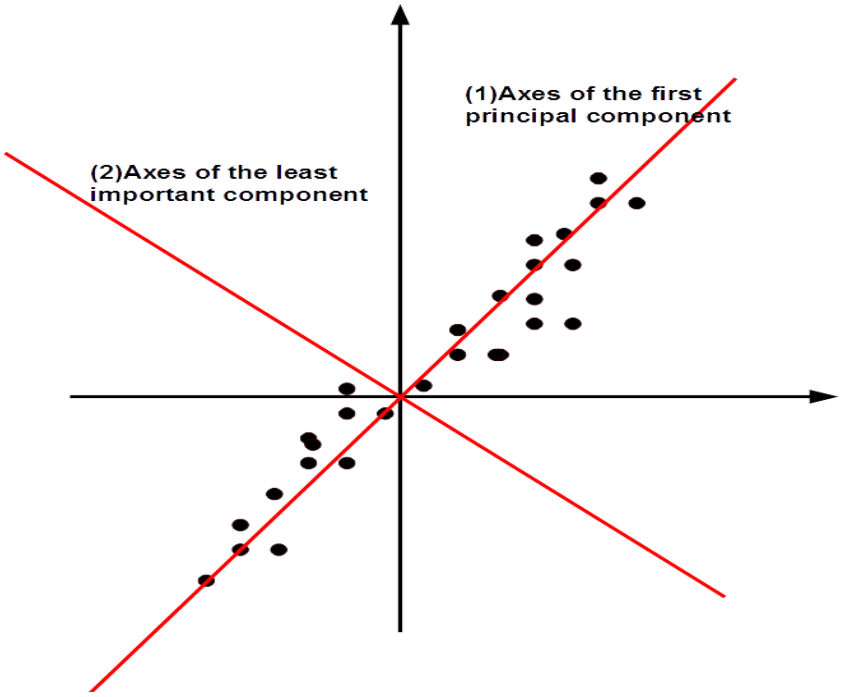


Figure 2.3 2D example of the PCA method

2.4.2 Breiman's variable importance Algorithm

Variable importance can be used in order to reduce the number of features that the classifier must be considered. Starting with many features, you train the classifier and then find the importance of each feature relative to other features. Unimportant feature can then be discarded, eliminating unimportant features improve speed performance.
The Breiman's variable important algorithm is given as follows[6]:

1. Train the classifier on the training data set
2. Use the validation or test to determine the accuracy of the classifier
3. For every data point and a chosen feature, randomly choose a new value for that feature from among the values the feature has in the rest of data set.
4. Train the classifier on the altered set of training data and measure the accuracy of classification on the altered test or validation data set. If randomizing a feature hurts accuracy a lot, then that feature is very important. If randomizing a feature does not hurt accuracy much, then that feature is of little importance and is candidate of removal.
5. Restore the original test or validation data set and try the next feature until we are done. The result is an ordering of each feature by its importance.

2.5 How to Diagnose Machine Learning Problem

Getting machine learning to work well can be more of an art than a science. Here we have some rules of thumb regarding machine learning[1]: more data beats less data, and better features beat better algorithms. There are two common problems see Figure 2.4:

Under fit model: That mean model assumptions are too strong for the data, so the model won't fit well. The possible solutions are as follows:

- More features can help make a better fit
- Use a more powerful algorithm

Over fit model: That means the algorithm used has memorized the data including noise so that it can't generalized. Some solutions are given as follows:

- More training data can help smooth the model
- Fewer features can reduce overfitting
- Use less powerful algorithm

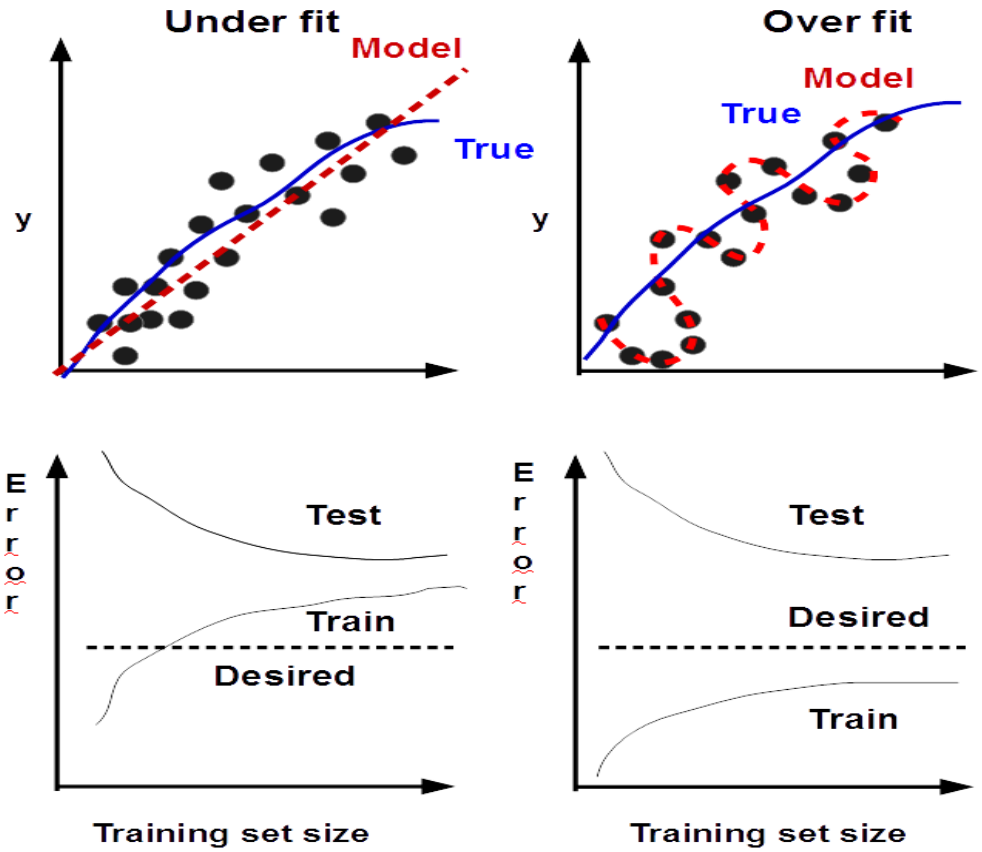


Figure 2.4 Pour model fitting in machine learning and its effect on training and test prediction performance.

Figure 2.4 shows under and overfitting of data on top and the corresponding error in term of the training set size. If we use the model that is to restrictive, we can never fit the true parabola in blue in the top left. The fit to both the training and the test data is poor. On the right side we fit the training data set exactly. It memorizes the training data as well as the noise in the data, the resulting fit to the test data is poor.

2.6 Cross-Validation, Bootstrapping, ROC and Confusion Matrix

There are some basic tools that are used in machine learning to measure results. In supervised learning, one of the most basic problems is simply knowing how well your algorithm has performed: how accurate is it at classifying or fitting data? In order to yield more accurate measures of actual performance of the machine learning algorithms, techniques such as cross-validation and/or bootstrapping are used [1].

Cross-validation involves dividing the data into K different sub-sets of data. The algorithm is trained on $K-1$ subset and test on the final subset of the data that wasn't trained on. This is done K times, where each of the K subsets gets a "turn" at being the validation set, and the average the result.

Bootstrapping is similar to cross-validation, but the validation set is selected at random from the training data.

Two other useful ways of assessing, characterizing, and tuning classifiers are plotting the receiver operating characteristic (ROC) and filling in a confusion matrix see Figure 2.5

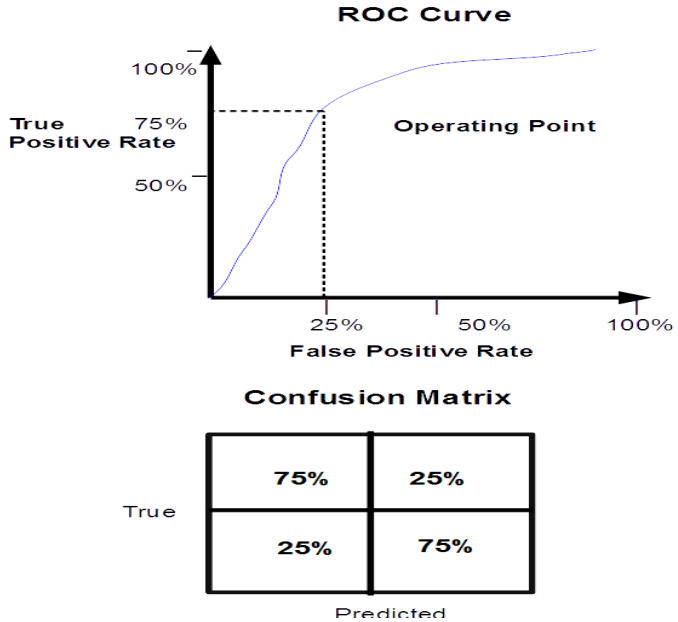


Figure 2.5 ROC curve and associated confusion matrix.

In Figure 2.5 the ROC curve measures the response of the classifier in term of performance parameters. The Figure 2.5 also shows a confusion matrix. This is just a chart of true and false positives along with true and false negatives.

3.0 Somme Common Algorithms

3.1 K-Means

K -means is a clustering algorithm. It attempts to find the natural cluster in the data. The K-means clustering algorithm is a particularly simple and effective approach to producing clusters on data. The idea is to represent each cluster by it's cluster center. Given cluster centers, we can simply assign each point to its nearest center. Similarly, if we know the assignment of points to clusters, we can compute the centers.

The following algorithm describes the K -means clustering algorithm in detail[10]. The cluster centers are initialized randomly. Data point x_n is compared against each cluster center μ_k . It is assigned to cluster k if k is the center with the smallest distance. The variable z_n stores the assignment (a value from 1 to K) of example n . The cluster centers are re-computed. First, X_k stores all examples that have been assigned to cluster k . The center of cluster k , μ_k is then computed as the mean of the points assigned to it. This process repeats until the means converge

Algorithm K-Means(D, K)

for $k = 1$ to K **do**

$\mu_k \leftarrow Some\ \ random\ \ location$ // randomly initialize mean for kth cluster

end for

repeat

for $n = 1$ **to** N **do**

$z_n \leftarrow \arg(\min_k \|\mu_k - x_n\|)$ // assign example n to closest center

end for

for $k = 1$ **to** K **do**

$X_k \leftarrow \{x_n : z_n = k\}$ // points assigned to cluster

$\mu_k \leftarrow Mean(X_k)$ // re-estimate mean of cluster k

end for

until μ_k stop changing

return z // return cluster assignments

3.2 K-Nearest Neighbors

The K-Nearest Neighbors is a classification technique. At training time, the entire training set is stored. At test time, a test example $\hat{x}$ is picked. To predict its label, we find the training example x that is most similar to $\hat{x}$. In particular, we find the training example x that minimizes $d(x, \hat{x})$ which is the measure of distance between x and $\hat{x}$ and is given as follows:

$$d(a,b) = \left[\sum_{i=1}^{D} (a_i - b_i)^2\right]^{1/2} \tag{3.1}$$

Where D is the number of data points. Since x is a training example, it has a corresponding label y. We predict that the label of $\hat{x}$ is also y.

Algorithm KNN-Predict($D, K, \hat{x}$)[10]

$S \leftarrow [\ ]$

for $n = 1$ **to** N **do**

$S \leftarrow S \otimes \langle d(x_n, \hat{x}), nn \rangle$ // store distance to training example

end for

$S \leftarrow sort(S)$ // put lowest-distance objects first

$\hat{y} \leftarrow 0$

for k = 1 to K do

$\langle dist, n \rangle \leftarrow S_k$ // n this is the kth closest data point

$\hat{y} \leftarrow \hat{y} + y_n$ // vote according to the label for the nth training point

end for

return $sign(\hat{y})$ // return $+1$ if $\hat{y} > 1$ and -1 if $\hat{y} < 1$

Despite its simplicity, this nearest neighbor classifier is incredibly effective. However, it is particularly prone to overfitting label noise.

3.3 Naïve Bayes Classifier

The Naïve Bases classifier some time called normal Bayes classifier is supervised classifier. It's naïve because it assumes that all the features are independent from one another even though this is seldom the case. Consider the probability denote p of an $Object$ given the $Feature_i$ see Figure 3.1, with $i = 1..5$

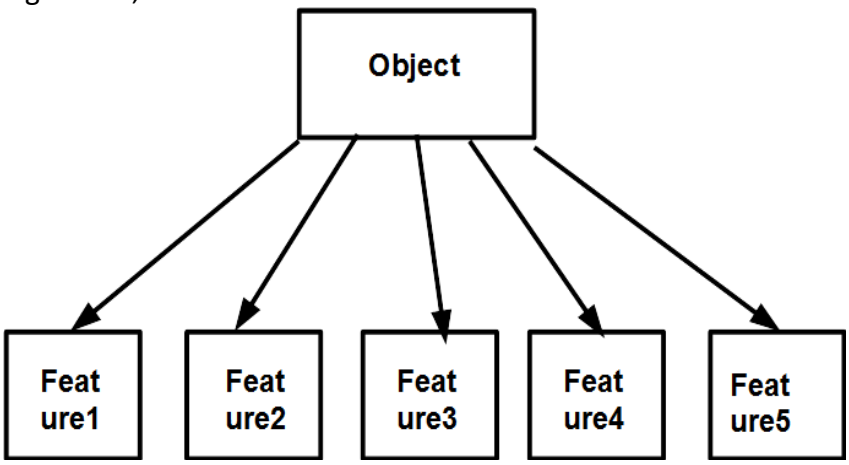


Figure 3.1 A naïve Bayesian network, where the lower-level features are caused by the presence of an object.

which can be written as follows using Bayes laws[1]:

$$\frac{p(Object \mid Feature1, Feature2, Feature3, Feature4, Feature5)}{p(Feature, Feature2, Feature3, Feature4, Feature5)} \tag{3.2}$$

We can use the definition of conditional probability to derive the joint probability

$$p(Object, Feature1, Feature2, Feature3, Feature4, Feature5) =$$
$$p(Object)\, p(Feature1 \mid Object)\, p(Feature2 \mid Object, Feature1)\, p(Feature3 \mid Object, Feature1, Feature2) \times$$
$$p(Feature4 \mid Object, Feature1, Feature2, Feature3)\, p(Feature5 \mid Object, Feature1, Feature2, Feature3, Feature4)$$

If we apply the assumption of independence of features, the conditional features drop out. We can therefore generalize as follows:

$$p(Object, all \quad Features) = p(Object) \prod_{i=1}^{all \quad Features} p(Feature_i \mid Object) \tag{3.3}$$

To use this as an overall classifier, we learn the models for the objects that we want. In the run mode we compute the features and find the objects that maximizes this equation. We typically test to see if the probability for this object is over a given threshold. If it is, then we declare the object to be found, if not, we declare that no object was recognized.

3.4 Expectation Maximization (EM) with Gaussian Mixture

Suppose that you have a probabilistic model that assumes access to labels, but you don't have any labels. You can treat the labels as hidden variables, and attempt to learn them at the same time as you learn the parameters of your model. There is a very broad family of algorithm for solving this kind of problem and is called Expectation Maximization family of algorithm.

Suppose that we are given a training set $\{x^{(1)}, x^{(2)}, ...x^{(m)}\}$ without labels. We wish to model the data by specifying a joint distribution The model $p(x^{(i)}, z^{(i)}) = p(x^{(i)} \mid z^{(i)})\, p(z^{(i)})$. Here $z^{(i)} \sim multinomid(\phi)$. Let k denote the number of values that the $z^{(i)}$ can take on. The problem can be described as follows each $x^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $\{1,2,.....k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depending on $z^{(i)}$ .This is called the mixture of Gaussians model. Note that here $z^{(i)}$'s are the hidden variables.

Define the likelihood data as follows:

$$l(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \log p(x^{(i)}; \phi, \mu, \Sigma)$$

$$= \sum_{i=1}^{m} \log \sum_{z^{(i)}=1}^{k} p(x^{(i)} \mid z^{(i)}; \phi, \mu, \Sigma)\, p(z^{(i)}; \phi)$$

Parameters ϕ, μ, Σ can be determined by setting the derivative of the likelihood function to zero. It can be shown that it is not possible to get maximum likelihood estimates of the parameters in closed form.

Note that if we knew what the $z^{(i)}$ were, the maximum likelihood problem would have been easy. Specifically, we could then write down the likelihood as

$$l(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \log p(x^{(i)} \mid z^{(i)}; \mu, \Sigma) + \log p(z^{(i)}; \phi)$$

Maximizing this with respect to ϕ, μ, Σ gives the parameters:

$$\phi_j = \frac{1}{m} \sum_{i=1}^{m} 1\{z^{(i)} = j\}$$

$$\mu_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}$$

$$\Sigma_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} x^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}$$

Note that 1{·} operator takes on a value of 1 if its argument is true, and 0 otherwise.

Since we don't have labels. One potential solution is to use iteration. We can start off with guesses for the values of the unknown variables, and then iteratively improve them over time An algorithm that does that is given as follows:

This EM algorithm is an iterative algorithm that has two main steps. In the E-step the values of the $z^{(i)}$'s are guessed. In the M-step, the parameters of our model are updated based on our guesses. Since in the M-step we are pretending that the guesses in the first part were correct, the maximization becomes easy. Here's the algorithm[7]:

Repeat until convergence: {

(E-step) For each i, j set

$$w^{(i)} := p(z^{(i)} = j \mid x^{(i)}; \phi, \mu, \Sigma)$$

(M-step) Update the parameters:

$$\phi_j = \frac{1}{m}\sum_{i=1}^{m} 1\{z^{(i)} = j\}$$

$$\mu_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}$$

$$\Sigma_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} x^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}$$

}

3.5 Support Vector Machine

3.5.1 Primal Optimization Problem

Let us consider the linear classification in two dimension. The straight line separates the positives from the negatives. It is defined by $w.x_i = t$, where w is a vector perpendicular to the decision boundary and pointing in the direction of the positives, t is the decision threshold, and x_i points to a point on the decision boundary. In particular, x_0 points in the same direction as w, from which it follows that $w.x_0 = \|w\|.\|x_0\| = t$ ($\|x\|$ denotes the length of the vector x).

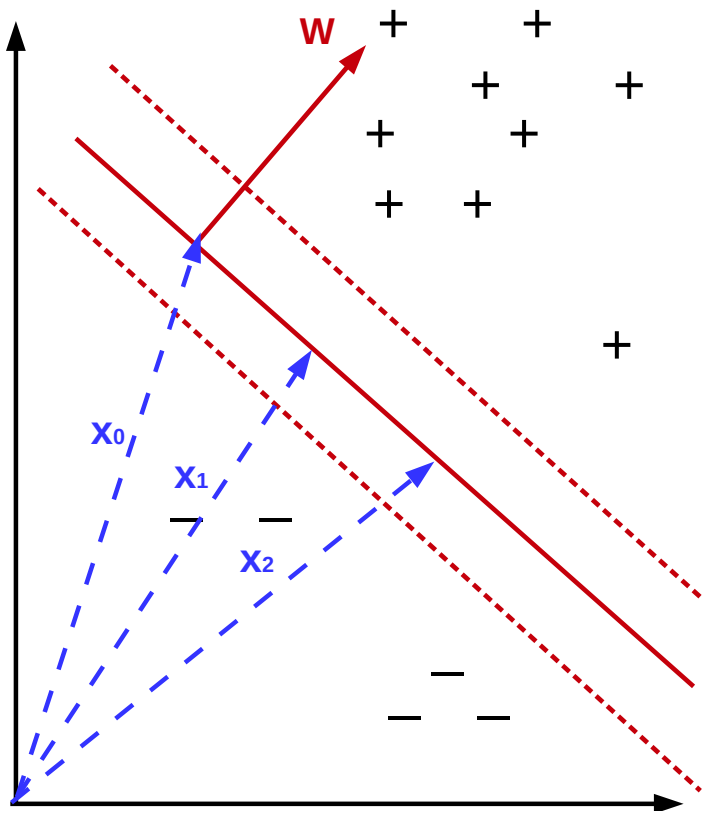


Figure 3.2 Linear classification in 2D

The geometry of a support vector classifier is given in Figure 3.3. The circled data points are the support vectors, which are the training examples nearest to the decision boundary. The support vector machine finds the decision boundary that maximizes the margin $m/\|w\|$.

Since we are free to rescale t, w and m, it is customary to choose $m=1$. Maximizing the margin then corresponds to minimizing $\|w\|$ or, more conveniently, $\frac{1}{2}\|w\|^2$, provided of course that none of the training points fall inside the margin.

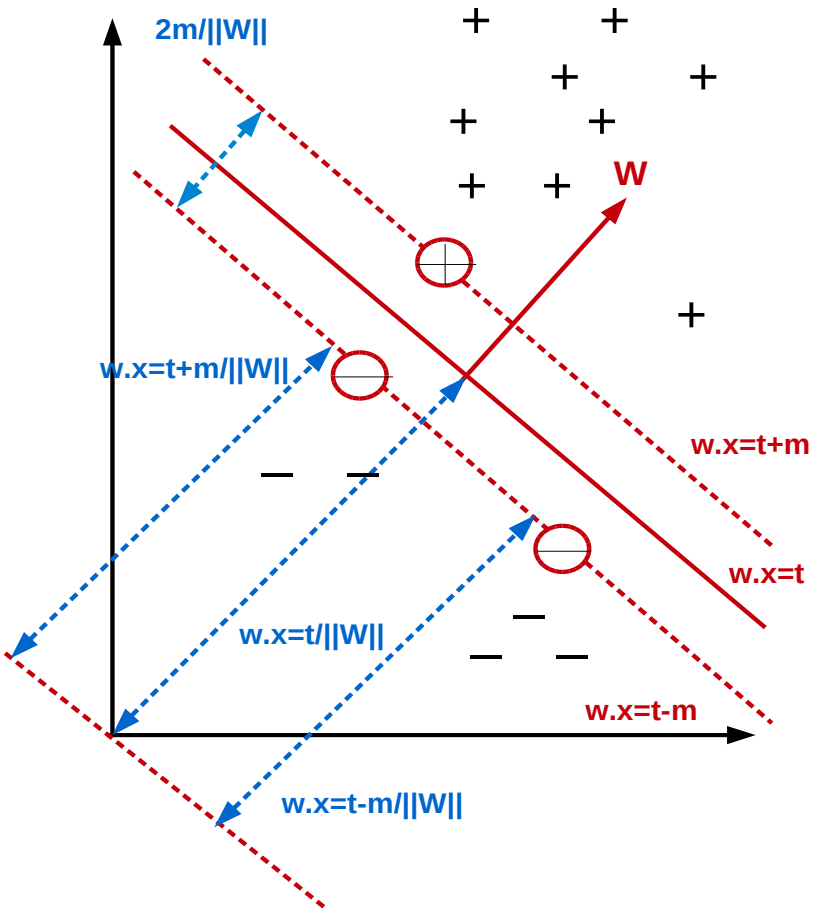


Figure 3.3 Geometry of the support Vector

This leads to a quadratic, constrained optimization problem[8]:

$$W^*, t^* = \arg\min_{w,t} \frac{1}{2}\|w\|^2 \qquad (3.5)$$

s.t $y_i(w.x_i - t) \geq 1, \leq i \leq n$

The above is an optimization problem with a convex quadratic objective function and only linear constraints. Its solution gives us the optimal margin classifier. This optimization problem can be solved using commercial quadratic programming (QP) code.

In the following, we will talk about Lagrange duality, because the dual form of this optimization problem will play a key role and will allow us to use kernels. Kernels are useful in the sens that they project data into higher dimensional space and then find the optimal linear separator between the classes.

3.5.2 Dual Optimization Problem

The dual optimization problem for support vector machines is to maximize the dual Lagrangian under positivity constraints and one equality constraint:

$$\alpha_1^*, \alpha_2^* .... \alpha_n^* = \arg\max_{\alpha_1^*, \alpha_2^* ... \alpha_n^*} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i . x_j + \sum_{i=1}^{n} \alpha_i \quad (3.6)$$

s.t $\alpha \geq 0, 1 \leq i \leq n$ and $\sum_{i=1}^{n} \alpha_i y_i$ =0

By examining the dual form of the optimization problem, we gained significant insight into the structure of the problem, and were also able to write the entire algorithm in terms of only inner products between input feature vectors. We will exploit this property to apply the kernels to our classification problem. The resulting algorithm, support vector machines, will be able to efficiently learn in very high dimensional spaces.

2.5.3 Kernel

Let us define a feature mapping ϕ the mapping of an attribute x into a feature space. Given a feature mapping ϕ, we define the corresponding Kernel to be

$$K(x, z) = \phi^T(x)\phi(z)$$

Now, given ϕ, we can compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. $K(x, z)$ is often inexpensive to calculate. This can be done by making SVMs to learn in the high dimensional feature space given by ϕ, without having to explicitly find or represent vectors $\phi(x)$ [8].

For example suppose we have $x, z \in R^n$ and consider $K(x, z) = (x^T z)^2$ We can also write this as $K(x, z) = \left(\sum_{i=1}^{n} x_i z_i \right) \left(\sum_{j=1}^{n} x_j z_j \right) = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i x_j z_i z_j = \sum_{i,j=1}^{n} (x_i x_j)(x_i x_j)$

Thus, we see that $K(x, z) = \phi^T(x)\phi(z)$, where the feature mapping ϕ is given (shown here for the case of n = 3) by

$$\phi(x) = [x_1 x_1, x_1 x_2, x_1 x_3, x_2 x_1, x_2 x_2, x_3 x_3, x_3 x_1, x_3 x_2, x_3 x_3]^T$$

Note that whereas calculating the high-dimensional $\phi(x)$ requires $O(n^2)$ time, finding $K(x, z)$ takes only $O(n)$ time—linear in the dimension of the input attributes.

Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ. Now, consider some finite set of m points (not necessarily the training set) $\{x^{(1)}, x^{(2)}, ... x^{(n)}\}$, and let a square, m-by-m matrix K be defined so that its (i, j)-entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the Kernel matrix.

Theorem (Mercer)[7]. Let $K : R^n \times R^n \mapsto R$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, x^{(1)}, ... x^{(m)}\}$, $(m < \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

3.5.4 Regularization and the non-separable case

The derivation of the SVM as presented so far assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some

cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outlier

To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization (using ℓ1 regularization) as follows:

$$\min_{\gamma,w,b} \|w\|^2 + \lambda \sum_{i=1}^{m} \xi_i \qquad (3.7)$$

$$\text{s.t} \quad y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1 - \xi_i \quad i = 1,...m$$

$$\xi_i \geq 0 \quad i = 1,...m$$

Thus, examples are now permitted to have (functional) margin less than 1, and if an example has functional margin $1-\xi_i$ (with $\xi_i > 0$), we would pay a cost of the objective function being increased by $\lambda\xi_i$. The parameter λ controls the relative weighting between the twin goals of making the $\|w\|^2$ small (which we saw earlier makes the margin large) and of ensuring that most examples have functional margin at least 1.

The dual of this problem is given as follows[7]

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^i, x^j \right\rangle \qquad (3.8)$$

$$0 \leq \alpha_i \leq \lambda \quad i = 1,..m$$

$$\text{s.t.} \quad \sum_{i=1}^{m} \alpha_i y^{(i)} = 0$$

3.5.5 The SMO algorithm

The SMO (sequential minimal optimization) algorithm, gives an efficient way of solving the dual problem arising from the derivation of the SVM. Let's talk about the coordinate ascent algorithm that can be useful in the solution of the SMO.

3.5.5.1 Coordinate ascent

Let consider the following unconstrained optimization problem

$$\max_{\alpha} W(\alpha_1, \alpha_2, ..\alpha_m) \qquad (3.10)$$

The solution of this problem involve the coordinate ascent algorithm that can be can be described as follows[8]:

Loop until convergence: {

For $i = 1,...m$

$$\arg\max_{\alpha} W(\alpha_1, \alpha_2, ..\alpha_{i-1,} \hat{\alpha}_i, \alpha_{i+1}, ...\alpha_m) \qquad (3.11)$$

}

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some α_i fixed, and reoptimize W with respect to just the parameter α_i.

3.5.5.2 SMO algorithm

Now we can use the SMO algorithm to solve dual problem as follows[7]:

Repeat till convergence {

1. Select some pair α_i and α_j to update next (using a heuristic that tries to pick the two that will allow us to make the biggest progress towards the global maximum).
2. Reoptimize $W(\alpha)$ with respect to α_i and α_j, while holding all the other α_k's $(k \neq i, j)$ fixed.

}

3.6 Decision Trees

Consider the problem of predicting a response or class y from inputs $x_1, x_2, \ldots x_p$ called feature. We do this by growing a binary tree. At each node in the tree, we apply a test to one of the inputs, say x_i. Depending on the outcome of the test, we go to either the left or the right sub-branch of the tree. Eventually we come to a leaf node, where we make a prediction. This prediction aggregates or averages all the training data points which reach that leaf.

Why do this? Predictors like linear or polynomial regression are global models, where a single predictive formula is supposed to hold over the entire data space. When the data has lots of features which interact in complicated, nonlinear ways, assembling a single global model can be very difficult, and hopelessly confusing when you do succeed. Some of the non-parametric smoothers try to fit models locally and then paste them together, but again they can be hard to interpret.

An alternative approach to nonlinear regression is to sub-divide, or partition, the space into smaller regions, where the interactions are more manageable. We then partition the sub-divisions again.

That's the recursive partition part; what about the simple local models? For classic regression trees, the model in each cell is just a constant estimate of y. That is, suppose the points $(x_1, y_1), (x_2, y_2) \ldots (x_c, y_c)$ are all the samples belonging to the leaf-node l. Then our model for l is just $\hat{y} = \frac{1}{c}\sum_{i=1}^{c} y_i$, the sample mean of the response variable in that cell. This is a piecewise-constant model. There are several advantages to this:

- Making predictions is fast (no complicated calculations, just looking up constants in the tree)
- It's easy to understand what variables are important in making the prediction (look at the tree)
- If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach

- The model gives a jagged response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves)
- There are fast, reliable algorithms to learn these trees

3.6.1 Entropy

Entropy is a measurement of chaos. It is a powerful tool that can be used in order to determine what features to use and how to carve up the feature space for achieving the best possible discrimination between the classes[2].

Entropy Definition: If a random variable X can take N different values, the ith value x_i with probability p_i, we can associate the following entropy with X :

$$H(X) = -\sum_{i=1}^{N} p(x_i)\log_2(x_i) \tag{3.12}$$

Conditional Entropy Definition: The conditional entropy $H(Y \mid X)$ measures how much entropy (chaos) remains in Y if we already know the value of the random variable X .

$$H(Y \mid X) = H(X,Y) - H(X) \tag{3.13}$$

Given two random variables X and Y , the entropy contained in both when taken together is $H(X,Y)$. The above definition says that, if X and Y are inter-dependent, and if we know X , we can reduce our measure of chaos in Y by the chaos that is attributable to X . $H(X,Y)$ is defined as follows:

$$H(X,Y) = -\sum_{i=1}^{N} p(x_i, y_j)\log_2(x_i, y_j) \tag{3.14}$$

Average Entropy Definition: Given N independent random variables $X_1, X_2 \ldots X_N$, we can associate an average entropy with all N variables by

$$H_{av} = -\sum_{i=1}^{N} H(X_i)p(X_i) \tag{3.15}$$

For another kind of an average, the conditional entropy $H(Y \mid X)$ is also an average, in the sense that the right hand side shown below is an average with respect to all of the different ways the conditioning variable can be instantiated:

$$H(Y \mid X) = -\sum_{i=1}^{N} H(Y \mid X = a)p(X = a) \tag{3.16}$$

3.6.2 Using Class Entropy to Discover the Best Feature for Discriminating Between the Classes

Let us consider the following set of features $x_1, x_2, \ldots . x_k$ we want to find out which of these feature is class discriminative?

To discover the best feature, all we have to do is to compute the class entropy as conditioned on each specific feature x separately as follows[2]

$$H(C \mid x) = -\sum_{i=1}^{N} H(C \mid v(x) = a)p(v(x) = a) \tag{3.17}$$

where the notation $v(x) = a$ means that the value of feature x is some specific value a. The we selects the feature f for which $H(C \mid x)$ is the smallest value

After finding the best feature for the root node in this manner, we can draw two branches from it, one for the training samples for which $v(x) \leq a$ and the other for the samples for which $v(x) > a$ as shown in the figure below:

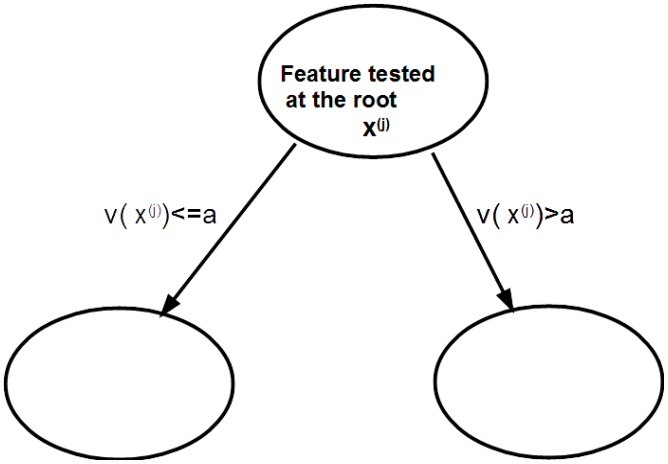


Figure 3.4 Example of binary decision tree.

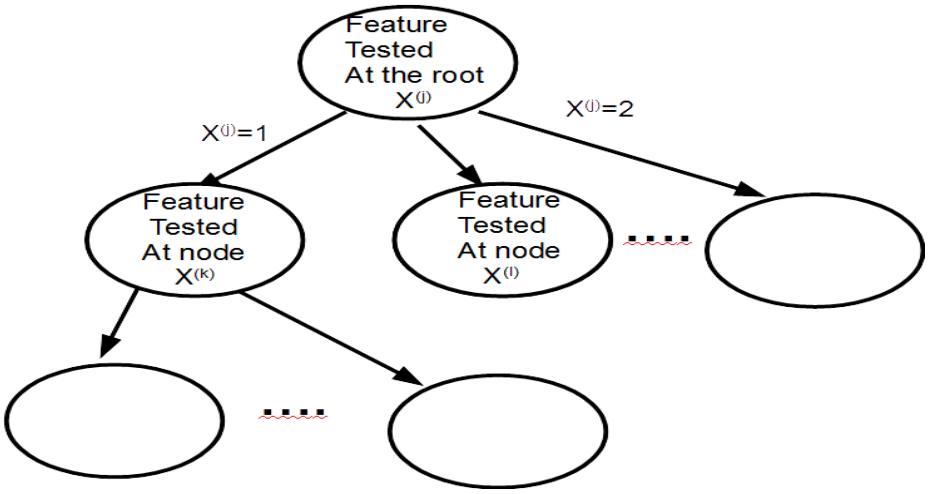


Figure 3.5 Example of arbitrary decision tree

Tree growing rule:

- You will add other child nodes to the root in the same manner, with one child node for each value that can be taken by the feature $x^{(j)}$.
- This process can be continued to extend the tree further to result in a structure that will look like what is shown in the Figure [3.4,3.5] above.
- A node is assigned the entropy that resulted in its creation. For example, the root gets the entropy $H(C)$ computed from the class priors.
- The children of the root are assigned the entropy $H(C \mid x^{(j)})$ that result in their creation

- A child node of the root that is on the branch $v(x^{(j)}) = a_j$ gets its own feature test (and is split further) if and only if we can find a feature $x^{(k)}$ such that $H(C \mid x^{(k)}, v(x^{(j)} = a_j)$ is less than the entropy $H(C \mid x^{(j)})$ inherited by the child from the root.
- If the condition $H(C \mid x^{(k)}, v(x^{(j)} = a_j) < H(C \mid x^{(j)})$ cannot be satisfied at the child node on the branch $v(x^{(j)}) = a_j$ of the root for any feature $x \neq x^{(j)}$ the child node remains without a feature test and becomes a leaf node of the decision tree.

Beside the entropy other metric relative to the data in every node of the tree is used. The term impurity is some time used to designate these metric.

3.6.3 Regression Impurity

For regression or function fitting, the equation for the node impurity is simply the square of the difference in value between the node value y and the data value x. We want to minimize[1]:

$$i(N) = \sum_j (y - x)^2 \qquad (3.18)$$

3.6.3 Classification Impurity

For classification, decision tree often use one of the three methods: entropy impurity, Gini impurity or misclassification impurity[1]. For these methods, we use the notation $P(C_j)$ to denote the fraction of pattern at the node N that are in class C_j each of these impurities has slightly different effects on the splitting decision. Gini is the most commonly used.

Entropy impurity

$$i(N) = -\sum_j P(C_j) \log_2 P(C_j) \qquad (3.19)$$

Gini impurity

$$i(N) = \sum_{j \neq i} P(C_j) P(C_i) \qquad (3.20)$$

Misclassification impurity

$$i(N) = 1 - \max P(C_j) \qquad (3.21)$$

Decision tree are the most widely used classification technology. This is due to their simplicity of implementation, ease of interpretation of result, flexibility with different data types.

3.7 Boosting

Decision tree are useful but they are often not the most performing classifier. Boosting use decision tree in their inner loop. Boosting algorithm, Adaboost as described here, are used to train N weak classifiers $h_n \in \{1,2.....n\}$. These classifiers are generally simple individually. In most case these classifiers are decision trees with only on split (called decision stumps). Each classifier is assigned a weighted vote v_n in final decision making process. We use a labeled data set of input feature vector $x^{(i)}$ each with scalar vector $y^{(i)}$ (where $i = 1,...M$ data point). For

Adaboost the label is binary $y^{(i)} \in [-1,1]$. We initialize a data point weighting distribution $D_n(i)$ that tells the algorithm how much misclassifying a data will cost. The algorithm is described as follows[1]:

1. $D_n(i) = \frac{1}{m}, i = 1,...m$
2. For $1 = 1,..n$:
    a. Find the classifier h_n that minimizes the $D_n(i)$ weighted error:
    b. $h_n = \arg\min_{h_j \in H} \varepsilon_j$ where $\varepsilon_n = \sum_{i=1}^{m} D_n(i)$ (for $y^{(i)} \neq h_j(x^{(i)})$) as long as $\varepsilon_j < 0.5$ else quit
    c. Set the voting weight $v_n = \frac{1}{2}\log[(1-\varepsilon_j)/\varepsilon_j]$ where ε_n is the arg min error from step 2b
    d. Update the data point weights: $D_{n+1}(i) = \frac{[D_n(i)\exp(-v_n y^{(i)} h_n(x^{(i)}))]}{Z_n}$ where Z_n normalize the equation over all data points i

Note that, in step 2b, if can't find a classifier with less than a 50% error rate then we quit; we probably need better features.
When the training algorithm just described is finished, the final strong classifier takes a new input vector x and classifies it using a weighted sum over the learned weak classifier h_n :

$$H(x) = sign\left(\sum_{n=1}^{N} v_n h_n(x)\right) \quad (3.22)$$

3.8 Random Trees

Random trees can learn more than one class at the same time simply by collecting the class votes at the leaves of each of many trees and selecting the class receiving the maximum votes as the winner[1]. Regression is done by averaging the values across the leaves of the forest. Random trees consist of randomly perturbed decision trees. Random trees also have the potential for parallel implementation, even on non-shared memory systems.
The basic system on which random trees are built is once again a decision tree. This decision tree is built all the way down until it's pure. Random tree cause each tree to be different by randomly selecting a different feature subset of the total feature from which the tree may learn at each node.

3.9 Multilayer Perceptron; Backpropagation.

The neuron is the fundamental cellular unit of the nervous systems and, in particular the brain. Each neuron is a simple micro-processing unit which receives and combines signals for many other neurons through input processes called dendrites[11]. If the combine signal is strong enough, it activate the firing of the neuron, which produces an output signal; the path of the

output signal is along a component of a cell call axon. This simple transfer of information is chemical in nature, but it has electrical side effects which can measure.

3.9.1 Neural Networks

In an artificial neural network, the unit analogous to the biological neuron is referred to as a "processing element" A processing element has many input paths (dendrites) and combines, usually by a simple summation, the values of these input paths. The result is an internal activity level for the processing element. The combine input is then modified by a transfer function. This transfer function can be a threshold function which only passes information if the combined activity reach a certain level, or it can be a continuous function of the combine input.

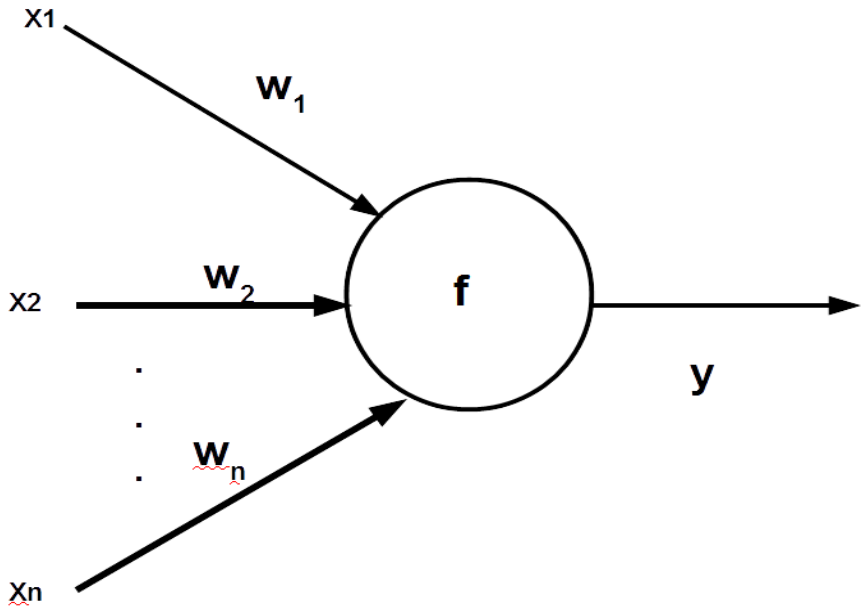


Figure 3.6 Schematic of an artificial neuron

The output path of a processing element can be connected to input paths of other processing elements through connection weights which correspond to the synaptic strength of neural connections. Since each connection has a corresponding weights, the signals on the input lines to a processing element are modified by these weights prior to being summed. The summation function is a weighted summation.

Processing elements are usually organized into groups called layers. A typical network consists of a sequence of layers with full or random connection between successive layers. There are typically two layers with connections to the outside world: An input buffer where data is presented to the network, and output buffer which holds the response of the network to a given input. Layers distinct from the input and output buffers are called hidden layers.

Sigmoid, Tanh, and ReLU Activation function

There are three major types of activation function that are used in practice that introduce nonlinearities in their computations:

Sigmoid. The first of these is the sigmoid neuron, which uses the function

$$f(z) = \frac{1}{1+e^{-z}} \qquad (3.23)$$

Intuitively, this means that when the logit or weighted sum of inputs is very small, the output of a logistic neuron is very close to 0. When the logit is very large, the output of the logistic neuron is close to 1

Tanh. The tanh use a similar kind of s-shaped nonlinearity, but instead of ranging from 0 to 1, the output of tanh neurons range from -1 to 1. As one would expect, they use

$$f(z) = \tanh(z) \qquad (3.24)$$

Restrict Linear Unit (ReLU). A different kind of nonlinearity is used by the restricted linear unit (ReLU) neuron. It uses the function

$$f(z) = \max(0, z) \qquad (3.25)$$

That results in a characteristic hockey stick shaped.

3.9.2 Softmax Output Layers

There are times when we want our output vector to be a probability distribution over a set of mutually exclusive labels. Using a probability distribution gives us a better idea of how confident we are in our predictions. The output vector in this case has the following form $[p_0, p_1, \ldots\ldots p_{n-1}]$ with n the number of the processing element in the output layer.

And $\sum_{i=0}^{n-1} p_i = 1$ (3.26)

This is achieved by using a special output layer called a softmax layer. Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all of the other neurons in its layer. This is because we require the sum of all the outputs to be equal to 1. Letting z_i be the logit of the ith softmax neuron, we can achieve this normalization by setting its output to[11]:

$$y_j = \frac{e^{z_j}}{\sum_{i=0}^{n-1} e^i} \qquad (3.27)$$

A strong prediction would have a single entry in the vector close to 1 while the remaining entries were close to 0.

3.9.3 Network Operation

There are two main phases in the operation of a network, learning and recall. In the most network these are distinct.

Learning is a process of adapting or modifying the connection weights in response to stimuli being presented at the input buffer. A stimulus presented at the output buffer corresponds to a desired response to a given input; this desired response must be provided by a knowledgeable teacher. In such a case the learning is referred to as “supervise learning”.

If the desired output is different from the input, the trained networks is referred to as hetero-associative network. If, for all training examples, the desired output vector is equal to the input

vector, the trained network is called auto-associative. If no desired output is shown, the learning is called unsupervised learning.

A third kind of learning, falling between supervised and unsupervised learning, is reinforcement learning where an external teacher indicates only whether the response to an input is good or bad. In some instances, the network may only be graded after several input have been processed by the network.

Whatever kind of learning is used, an essential characteristic of any network is its learning rule. The learning rule specifies how weights adapt in response to a learning example.

3.9.4 MLP learning rule

The multilayer perceptron is a neural network that is comprised of an input layer , an output layer and at least on hidden layer. There is no theoretical limit on the number of hidden layers. These network are also called feedforward networks. They operate by feeding data forwards along the interconnections from input layer, through the hidden layer to the output layer. The multilayer perceptron is a neural network that still ranks among top-performing classifiers, especially for text recognition. It can be rather slow in training because it uses gradient descent to minimize error by adjusting weighted connections between the numerical classification nodes within the layers. In the test mode it is quite fast.

The following notation are used in order to described the learning rule.

$x_j^{[s]}$: the current output state of the jth neuron in layer s

$w_{ji}^{[s]}$: weight on connection joining ith neuron in layer $s-1$ to jth in layer s

$I_j^{[s]}$: weighted summation of inputs to jth neuron in layer s

A back-propagation element therefore transfers its input as follows:

$$x_j^{[s]} = f\left(\sum_i \left(w_{ji}^{[s]}.x_i^{[s-1]}\right)\right) \quad (3.28)$$

$$= f\left(I_j^{[s]}\right)$$

Where f is traditionally the sigmoid function but can be any differentiable function. The sigmoid function is defined as

$$f(z) = \frac{1}{1+e^{-z}}$$

Backpropagating the local error

Suppose that the network has some global error function E associated with it which is a differentiable function of all the connection weights in the network. The actual error function is unimportant to understand the mechanism of back-propagation. The critical parameter that is passed back through the layers is defined by

$$e_j^{[s]} = \partial E / \partial I_j^{[s]} \quad (3.29)$$

Using the chain rule twice in succession gives the relationship between the local error at a particular processing element at level s and all the local errors at the level $s+1$

$e_j^{[s]} = f'(I_j^{[s]})\sum_k \left(e_k^{[s+1]}.w_{kj}^{[s+1]}\right)$ (3.30)

If f is the sigmoid function as defined , then its derivative can be expressed as a simple function of itself as follows:

$f'(z) = f(z)(1-f(z))$ (3.31)

Therefore after combining the local error can be written as follows[11]:

$e_j^{[s]} = x_j^{[s]}(1-x_j^{[s]})\sum_k \left(e_k^{[s+1]} w_{kj}^{[s+1]}\right)$ (3.32)

Minimizing the Global error

Given the current set of weights $w_{ij}^{[s]}$, we need to determine how to increment or decrement them in order to decrease the global error.

$\Delta w_{ji}^{[s]} = -\eta\, \partial E / \partial w_{ji}^{[s]}$ (3.33)

Where η is the learning rate. The partial derivative can be calculated from the local error.

$$\partial E/\partial w_{ji}^{[s]} = \left(\partial E/\partial I_j^{[s]}\right)\left(\partial I_j^{[s]}/\partial w_{ji}^{[s]}\right) = -e_j^{[s]} x_i^{[s-1]} \quad (3.34)$$

After combining the two previous equation this gives

$\Delta w_{ji}^{[s]} = -\eta e_j^{[s]} x_i^{[s-1]}$ (3.35)

The Global Error function

Suppose that a vector I is presented at the input edge layer of the network and suppose the desired output d is specified. Let denote the actual output produced by the network with its current set of weights. Then a measure of the error in achieving that desired output is given by

$E = \frac{1}{2}\sum_k \left((d_k - o_k)^2\right)$ (3.36)

The scale local error for each element of the output layer is given as follows[11]:

$e_k^0 = -\partial E/\partial I_k^0 = -\partial E/\partial o_k \,.\partial o_k/\partial I_k = (d_k - o_k) f'(I_k)$ (3.37)

Summary of the Standard Back-Propagation Algorithm

Given an input vector I and desired output vector d do the following:

1. Present I to the input layer of the network and propagate it through to the output to obtain an output vector o.
2. As this information propagates through the network, it will also set all the summed input I_j and output states x_j for each processing element in the network.
3. For each processing element in the output layer, calculate the scaled local error as given in [3.37] and then calculate the delta weight using [3.35]

4. For each layer s, starting at the layer below the output layer and ending with the layer above the input, and for each processing element in the layer s, calculate the scaled local error as given in [3.32] then calculate the delta rule using [3.35]
5. Update all weights in the network by adding the delta weights to the corresponding previous weights.

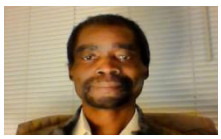
4.0 Conclusion

This tutorial is an overview of the most popular machine learning algorithms. The part two of this tutorial will be on deep learning.

5.0 References

[1] Bradski Gary, Adrian Kaeller "OpenCV, computer Vision with OpenCV Library"
O'reilly, September 2008: First Edition
[2] Kak Avinash " Decision Trees: How to construct them ad How to use them for classifying new data"
Perdu university, 2016
[3] Wei-yin Loh " Classification and regression Trees"
Wiley & Sons, inc, Wires Data Mining Knowledge Discovery 2011, 1, 24-23
[4] Flack Peter A. "Machine learning the Art and Science of Algorithms that Make Sense of Data"
Intelligent System Laboratory, university of Bristol, united kingdom, December 29, 2013
[5] Hastie Trevor "Tree, Bagging, Random Forest and Boosting"
https://web.stanford.edu/~hastie/pub.htm
[6] Breiman, Leo " Decision Trees and Random Forests"
http://www.stat.berkely.edu/~breiman/randomForest
[7] Andrew Ng "The EM algorithm - CS229"*cs229.stanford.edu/notes/cs229-notes8.pdf*
[8] Andrew Ng. "Support Vector Machines - CS229"
cs229.stanford.edu/notes/cs229-notes3.pdf
[10] Dumee III Hall, "A course in Machine Learning"
Version 8 August 2012: http://ciml.info
[11] Technical Publication Group, "Neural Computing"
NeuralWare, Inc., 1993

About the author:



Sylvain Dindy-Bolongo is currently Principal at VizCortex, prior to that, he worked as Software Engineer, Embedded Software Engineer, Control System Engineer, Control Architect for several companies in the San Francisco Bay Area, including HP, Agilent, Schneider Electric, Tech-Mahindra, as well as a few startups. He also worked as Part-Time Instructor for

the Art Institute of California at San Francisco, University of California at Bekeley Extension, Sonoma State University (Master of Engineering Program).
His interests include Robotics (SLAM Navigation, Manipulator Control), Machine Learning, Deep Learning, Computer Vision, Control Systems (Robust, Intelligent, Embedded).
He earned his BEE, MSEE from Ecole Polytechnique de Montreal (University of Montreal, Canada) and a PhD in Control Systems and Robotics from Wichita State University, Kansas.