Assignment

B ANIRUDH SRINIVASAN
COE17B019

1. Decision Tree

1.1 Introduction

Decision Tree Mining is a type of data mining technique that is used to build Classification Models. It builds classification models in the form of a tree-like structure. This type of mining belongs to supervised class learning. Decision trees can be used for both categorical and numerical data.

Decision Tree has three parts,

- An inner node which represents an attribute.
- An edge representing the test on the father node.
- And a leaf node representing one of the classes.

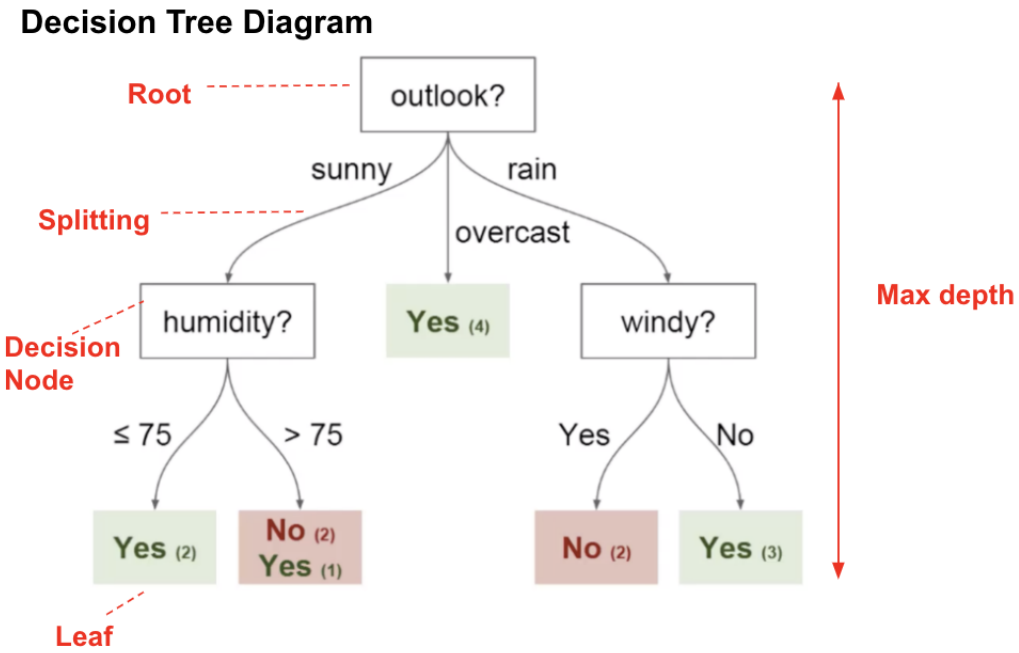


Figure 1: figure explaining different part of a decision tree

1.2 Construction of a decision tree with example

Example Dataset

Attributes				Classes
Outlook	Temperature	Humidity	Windy	Play Golf
Rainy	Hot	High	FALSE	No
Rainy	Hot	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Sunny	Mild	High	FALSE	Yes
Sunny	Cool	Normal	FALSE	Yes
Sunny	Cool	Normal	TRUE	No
Overcast	Cool	Normal	TRUE	Yes
Rainy	Mild	High	FALSE	No
Rainy	Cool	Normal	FALSE	Yes
Sunny	Mild	Normal	FALSE	Yes
Rainy	Mild	Normal	TRUE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Sunny	Mild	High	TRUE	No

Figure 2: Example dataset

Step 1: Determine the Decision Column

Since decision trees are used for classification, you need to determine the classes which are the basis for the decision.

In this case, it it the last column, that is Play Golf column with classes Yes and No.

To determine the rootNode we need to compute the entropy. To do this, we create a frequency table for the classes (the Yes/No column).

Play Golf(14)	
Yes	No
9	5

Figure 3: Frequency table for Play Golf attribute

Step 2: Calculating Entropy for the classes

In this step, you need to calculate the entropy for the Decision column Formula for entropy is :

$$Entropy(Attribute) = -\sum_{i \in c} p_i \log_2 p_i \tag{1}$$

where,

- c is set of all classes for the attribute.
- p_i is the probability associated with that class.

The calculation of Play Golf attribute is :

$$E(PlayGolf) = E(5,9)$$
$$= -\left(\frac{9}{14}\log_2\frac{9}{14}\right) - \left(\frac{5}{14}\log_2\frac{5}{14}\right)$$
$$= -(0.357\ \log_2 0.357) - (0.643\ \log_2 0.643)$$
$$= 0.94$$

Figure 4: Calculating Entropy

Step 3: Calculate Entropy for Other Attributes After Split

We need to calculate the entropy of the resultant tree, wrt each attribute and use it to calculate *Gain*. The entropy for two variables is :

$$Entropy(A1, A2) = \sum_{i \in c} P(i)E(i) \tag{2}$$

where,

- c is set of all classes for the attribute A2.
- P(i) is the probability associated with the i^th class of attribute A2.
- E(i^th) is the entropy of the i^th class of attribute A2.

The calculation of Play Golf attribute is :
E(PlayGolf, Outloook)
E(PlayGolf, Temperature)
E(PlayGolf, Humidity)
E(PlayGolf,windy)

		PlayGolf(14)		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5

Figure 5: Table for Outlook and Play Golf

E(PlayGolf, Outlook) = P(Sunny) E(Sunny) + P(Overcast) E(Overcast) + P(Rainy) E(Rainy)

Which is same as:

E(PlayGolf, Outlook) = P(Sunny) E(3,2) + P(Overcast) E(4,0) + P(rainy) E(2,3)

E(x,y) = E(y,x)
Therefore, E(2,3) = E(3,2)

$$E(Sunny) = E(3,2)$$

$$= -\left(\frac{3}{5}\log_2\frac{3}{5}\right) - \left(\frac{2}{5}\log_2\frac{2}{5}\right)$$

$$= -(0.60\ \log_2 0.60) - (0.40\log_2 0.40)$$

$$= -(0.60 * 0.737) - (0.40 * 0.529)$$

$$= \mathbf{0.971}$$

Figure 6: Calculating E(3, 2)

$$E(Overcast) = E(4,0)$$

$$= -\left(\frac{4}{4}\log_2\frac{4}{4}\right) - \left(\frac{0}{4}\log_2\frac{0}{4}\right)$$

$$= -(0) - (0)$$

$$= \mathbf{0}$$

Figure 7: Calculating E(4, 0)

Therefore, the E(Play Golf, Outlook) is :

$$E(PlayGolf, Outlook) = \frac{5}{14}E(3,2) + \frac{4}{14}E(4,0) + \frac{5}{14}E(2,3)$$

$$= \frac{5}{14}0.971 + \frac{4}{14}0.0 + \frac{5}{14}0.971$$

$$= 0.357 * 0.971 + 0.0 + 0.357 * 0.971$$

$$= \mathbf{0.693}$$

Figure 8: Calculating Entropy

Similarly,

		PlayGolf(14)		
		Yes	No	
Temperature	Hot	2	2	4
	Cold	3	1	4
	Mild	4	2	6

Figure 9: Table for Temperature and Play Golf

E (PlayGolf, Temperature) = **4/14 * E**(Hot) + **4/14 * E**(Cold) + **6/14 * E**(Mild)

E (PlayGolf, Temperature) = **4/14 * E**(2, 2) + **4/14 * E**(3, 1) + **6/14 * E**(4, 2)

E (PlayGolf, Temperature) = 4/14 * -(2/4 log 2/4) – (2/4 log 2/4)
+ 4/14 * -(3/4 log 3/4) – (1/4 log 1/4)
+ 6/14 * -(4/6 log 4/6) – (2/6 log 2/6)

E (PlayGolf, Temperature) = 5/14 * 1.0
+ 4/14 * 1.811
+ 5/14 * 0.918
= 0.911

Figure 10: Calculating E(Play Golf, Temperature)

		PlayGolf(14)		
		Yes	No	
Humidity	High	3	4	7
	Normal	6	1	7

Figure 11: Table for Humidity and Play Golf

$$E(\text{PlayGolf, Humidity}) = \mathbf{7/14} * \mathbf{E}(\text{High}) + \mathbf{7/14} * \mathbf{E}(\text{Normal})$$

$$E(\text{PlayGolf, Humidity}) = \mathbf{7/14} * \mathbf{E}(3, 2) + \mathbf{7/14} * \mathbf{E}(4, 0)$$

$$E(\text{PlayGolf, Humidity}) = 7/14 * -(3/7 \log 3/7) - (4/7 \log 4/7)$$
$$+ 7/14 * -(6/7 \log 6/7) - (1/7 \log 1/7)$$

$$E(\text{PlayGolf, Humidity}) = 7/14 * 0.985$$
$$+ 7/14 * 0.592$$
$$= \mathbf{0.788}$$

Figure 12: Calculating E(Play Golf, Humidity)

		PlayGolf(14)		
		Yes	No	
Windy	TRUE	3	3	6
	FALSE	6	2	8

Figure 13: Table for windy and Play Golf

$$E(\text{PlayGolf, Windy}) = \mathbf{6/14} * \mathbf{E}(\text{True}) + \mathbf{8/14} * \mathbf{E}(\text{False})$$

$$E(\text{PlayGolf, Windy}) = \mathbf{6/14} * \mathbf{E}(3, 3) + \mathbf{8/14} * \mathbf{E}(6, 2)$$

$$E(\text{PlayGolf, Windy}) = 6/14 * -(3/6 \log 3/6) - (3/6 \log 3/6) + 8/14 * -(6/8 \log 6/8) - (2/8 \log 2/8)$$

$$E(\text{PlayGolf, Windy}) = 6/14 * 1.0 + 8/14 * 0.811 = \mathbf{0.892}$$

Figure 14: Calculating E(Play Golf, y)

So now that we have all the entropies for all the four attributes, let's go ahead to summarize them as shown in below:

E(PlayGolf, Outloook) = 0.693
E(PlayGolf, Temperature) = 0.911
E(PlayGolf, Humidity) = 0.788
E(PlayGolf,windy) = 0.892

Step 4: Calculating Information Gain for Each Split

The next step is to calculate the information gain for each of the attributes. The information gain is calculated from the split using each of the attributes. Then the attribute with the largest information gain is used for the split.

The information gain is calculated using the formula:

$$Gain(S, T) = Entropy(S) – Entropy(S, T) \tag{3}$$

where,

- Entropy(S) is the entropy associated with attribute S.
- Entropy(S, T) is the joint entropy of attributes, S and T.

For example, the information gain after splitting using the Outlook attribute is given by:

Gain(PlayGolf, Outlook) = Entropy(PlayGolf) – Entropy(PlayGolf, Outlook)
= 0.94 – 0.693
= 0.247

Gain(PlayGolf, Temperature) = Entropy(PlayGolf) – Entropy(PlayGolf, Temparature)
= 0.94 – 0.911
= 0.029

Gain(PlayGolf, Humidity) = Entropy(PlayGolf) – Entropy(PlayGolf, Humidity)
= 0.94 – 0.788
= 0.152

Gain(PlayGolf, windy) = Entropy(PlayGolf) – Entropy(PlayGolf, windy)
= 0.94 – 0.892
= 0.048

Having calculated all the information gain, we now choose the attribute that gives the highest information gain after the split.

Step 5: Perform the First Split

Now that we have all the information gain, we then split the tree based on the attribute with the highest information gain.

From our calculation, the highest information gain comes from Outlook. Therefore the split will look like this:

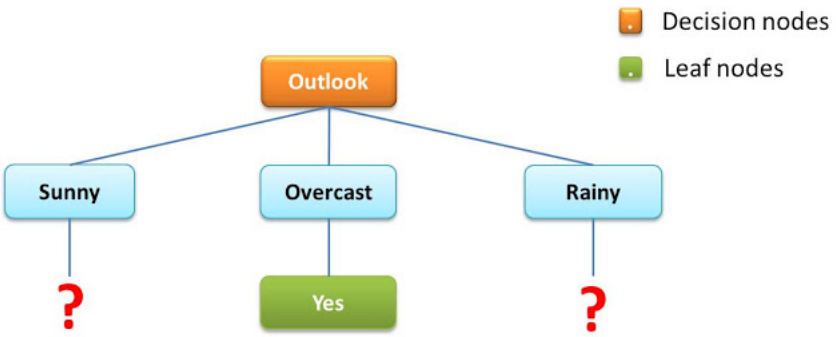


Figure 15: Decision Tree after first split

Now that we have the first stage of the decison tree, we see that we have one leaf node. But we still need to split the tree further.

To do that, we need to also split the original table to create sub tables. This sub tables are given in below.

Outlook	Temperature	Humidity	Windy	Play Golf
Sunny	Mild	Normal	FALSE	Yes
Sunny	Mild	High	FALSE	Yes
Sunny	Cool	Normal	FALSE	Yes
Sunny	Cool	Normal	TRUE	No
Sunny	Mild	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Overcast	Cool	Normal	TRUE	Yes
Rainy	Hot	High	FALSE	No
Rainy	Hot	High	TRUE	No
Rainy	Mild	High	FALSE	No
Rainy	Cool	Normal	FALSE	Yes
Rainy	Mild	Normal	TRUE	Yes

Figure 16: Sub Tables

Step 6: Perform Further Splits

Now, we need to recalculate the entropies and the gain for the remaining attributes, and split with the attribute that has the maximum gain.

Note that, an attribute wont repeat in the subtree with that attribute as node.
Splitting for Rainy

From the above table its clear that Humidity gives an homogeneous split, i.e Entropy(Outlook = Rainy, Humidity) = 0
Therefore, lets split Rainy with Humidity

Similarly, when the Outlook is Sunny, the attribute windy gives an homogeneous split, thus we split Sunny with windy.

Step 7: Complete the Decision Tree

Keep on splitting the decision tree, until no other attribute is left, or if we get an homogeneous split.
Therefore, the final table is :

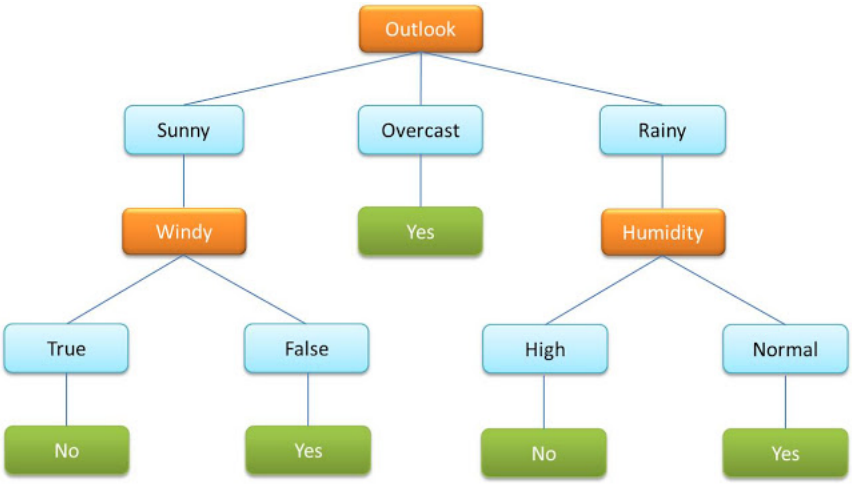


Figure 17: Sub Tables

1.3 Pseudo-Code for Decision Tree construction

```
Begin
Load learning sets  and create decision tree root node(rootNode), add learning
     set S into root not as its subset
For rootNode, compute Entropy(rootNode.subset) first
If Entropy(rootNode.subset) == 0 (subset is homogenious)
       return a leaf node

If Entropy(rootNode.subset)!= 0 (subset is not homogenious)
     compute Information Gain for each attribute left (not been used for
    spliting)
     Find attibute A with Maximum(Gain(S, A))
     Create child nodes for this root node and add to rootNode in the decision
     tree

For each child of the rootNode
   Apply ID3(S, A, V)
   Continue until a node with Entropy of 0 or a leaf node is reached
End
```

1.4 Additional Notes

1. For numerical attributes,

   Step 1: Sort the Attribute in ascending order.

   Step 2: Calculate the avg value for two adjacent datapoints.

   Step 3: Now consider each avg value to be a class, and calculate the Gain. Choose the average value with the most gain.

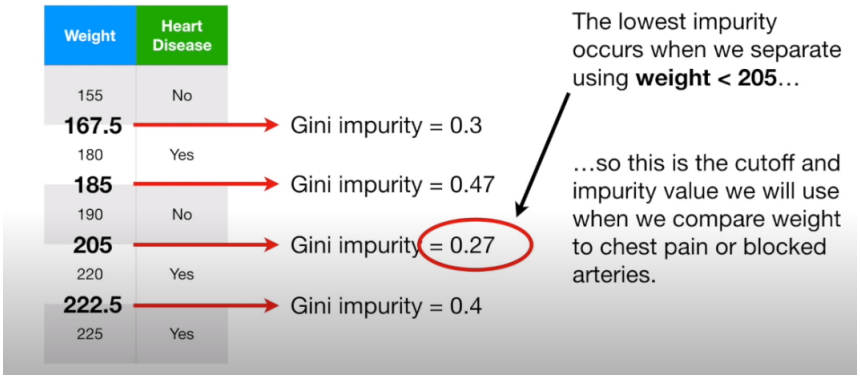


Figure 18: Illustration of the above steps. Note that here gini impurity is used and accordingly the one with the least impurity is chosen.

2. For attributes with rank as their value, we try to split based on each possible rank.

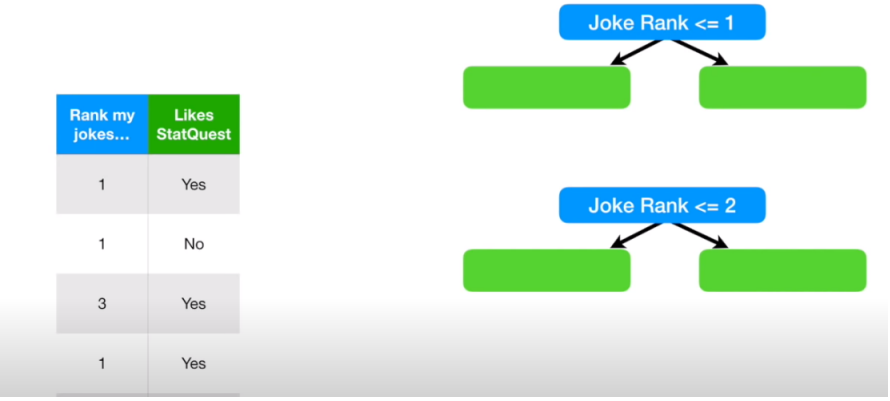


Figure 19: Illustration of the above steps. Note that the highest rank is avoided as all other ranks are less than it.

3. We can use other parameters to identify the attribute which splits the tree.
   The commonly used statistical measures are :

   - **Entropy and Information Gain**
     Entropy quantifies how much information there is in a random variable, or more specifically its probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy. Information gain is the reduction in entropy or surprise by transforming a dataset and is often used in training decision trees.

$$Entropy(Attribute) = -\sum_{i \in c} p_i \log_2 p_i \tag{4}$$

     where,

     – c is set of all classes for the attribute.
     – p_i is the probability associated with that class.

     And then we find the information gain, and choose the attribute with the highest gain.

$$Gain(S,T) = Entropy(S) – Entropy(S,T) \tag{5}$$

     where,

     – Entropy(S) is the entropy associated with attribute S.
     – Entropy(S, T) is the joint entropy of attributes, S and T.

- **Gini Impurity and Gini Gain**
  Gini Impurity is a measurement of the likelihood of an incorrect classification of a new instance of a random variable, if that new instance were randomly classified according to the distribution of class labels from the data set.

$$G(Attribute) = \sum_{i \in c} p_i * (1 - p_i) \tag{6}$$

  where,

  – c is set of all classes for the attribute.
  – p_i is the probability associated with that class.

  And then we find the gini gains, similar to information gain, and choose the attribute with the highest gain.

$$GG(Attribute) = 1 - Remainder_Impurity(attribute) \tag{7}$$

$$RemainderImpurity(attribute) = \sum_{i \in B} p_i * (G(i)) \tag{8}$$

  where,

  – B is the, set of all branches for the attribute.
  – p_i is the probability associated with that branch.

  It seems that gini impurity and entropy are often interchanged in the construction of decision trees. Neither metric results in a more accurate tree than the other.

  All things considered, a slight preference might go to gini since it doesn't involve a more computationally intensive log to calculate.

- **Gain Ratio**
  In decision tree learning, Information gain ratio is a ratio of information gain to the intrinsic information.

  Information gain ratio biases the decision tree against considering attributes with a large number of distinct values. So it solves the drawback of information gain—namely, information gain applied to attributes that can take on a large number of distinct values might learn the training set too well. For example, suppose that we are building a decision tree for some data describing a business's customers. Information gain is often used to decide which of the attributes are the most relevant, so they can be tested near the root of the tree. One of the input attributes might be the customer's credit card number. This attribute has a high information gain, because it uniquely identifies each customer, but we do not want to include it in the decision tree: deciding how to treat a customer based on their credit card number is unlikely to generalize to customers we haven't seen before.

$$IV(E_x, a) = -\sum_{v \in values(a)} \frac{|\{x \in E_x | value(x, a) = v\}|}{|E_x|} * log_2 \frac{|\{x \in E_x | value(x, a) = v\}|}{|E_x|} \tag{9}$$

where,

- IV is the Intrinsic Inforamtion.
- E_x is the dataset.
- value(x, a) returns the value of attribute a for the datapoint x.
- a A, where A is the set of all attributes of E_x.

$$GainRatio(S, A) = \frac{Gain(S, A)}{IV(S, A)}) \tag{10}$$

where,

- S is the Target Variable.
- A is an attribute.

2. Naive Bayes Classifier

2.1 Introduction

A Naive Bayes classifier is a probabilistic machine learning model that's used for classification task. The crux of the classifier is based on the Bayes theorem.

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{11}$$

where,

- P(A|B) is the posterior probability of class (A, target) given predictor (B, attributes).
- P(A) is the prior probability of class.
- P(B) is the likelihood which is the probability of predictor given class.
- P(B|A) is the likelihood which is the probability of predictor given class.

  The classification rule is simple, a datapoint is classified into the class with the maximum posteriror probability for that class.

2.2 Classifying using Naive Bayesian with example

Example Dataset

Table 1: Example dataset

Name	Age	Gender
Drew	14	M
Karin	20	F
Nina	55	F
Alberto	17	M
Drew	28	M
Alberto	43	F
Karin	72	F
Claudia	35	F
Drew	4	M
Sergio	57	F

Step 1: Calculating the probabilities

According to bayes theorem,

we have, X $\in$C$_i$ if $P(C_i|X)$ is maximum among all other posterior class probabilities.

$$P(C_i|X) = \frac{P(X|C_i) * P(C_i)}{P(X)} \tag{12}$$

P(X) is a constant for all classes, hence we can omit it.

If we don't know the class probabilities (i.e. P(C_i)), we can assume data to be equally distributed among the classes and omit it also. Another way is to define

$$P(C_i) = \frac{Number\ of\ tuple\ containing\ class\ C_i}{Total\ number\ of\ Tuples} \tag{13}$$

As, X is a n - dimensional vector, x_1, x_2, ..., x_n,

$$P(X|C_i) = \Pi_{k=1}^{n} P(x_i|C_i) \tag{14}$$

This is because, we have assumed that class conditional independence, meaning attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes).

If the attribute from which x_i originates is a categorical attribute,

then,

$$P(x_i|C_i) = \frac{Number\ of\ tuples\ in\ C_i\ containing\ x_i}{Total\ number\ of\ tuple\ in\ C_i} \tag{15}$$

If the attribute from which x_i originates, is a continually valued numerical attribute, then, we assume that this attribute follows a gaussian distribution.

$$P(x_i|C_i) = g(x_i, \mu_{C_i}, \sigma_{C_i}) \tag{16}$$

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-} \frac{(x-\mu)^2}{2\sigma^2} \tag{17}$$

where,

- $g(x, \mu, \sigma)$ is the gaussian distribution with mean μ and standard deviation σ. $\mu_{C_i} and \sigma_{C_i}$ are the mean and standard deviation of the values of the attribute of class C_i.

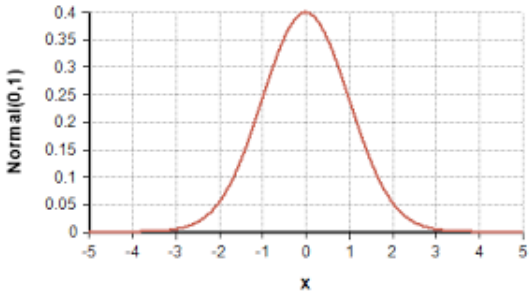


Figure 20: Standard Normal Gaussian Distribution

For our example,

Lets take the test vector = ("Alberto", 25)

We need to identify the gender attribute of the above test vector.

Let C_1 be F and C_2 be M.

From eq(13),

$$\mathrm{P}(C_1) = \frac{number\ of\ tuples\ in\ C_1}{total\ number\ of\ tuples}$$

$$= \frac{4}{10} = 0.4$$

Similarly,

$$\mathrm{P}(C_2) = 0.6$$

From eq(14),

$$P(X|C_1) = P(x_{name}|C_1) * P(x_{age}|C_1)$$

From eq(15),

$$P(x_{name} = "Alberto"|C_1) = \frac{Number\ of\ tuples\ in\ class\ 1\ with\ name\ attribute\ as\ "Alberto"}{Total\ number\ of\ tuples\ in\ class\ 1}$$

$$= \frac{1}{4} = 0.25$$

Similarly,

$$P(x_{name} = "Alberto"|C_2) = \frac{1}{6} = 0.1667$$

Now, age is an numerical attribute,

Hence we need to find the distribution.

From eq(16) and eq(17),

$$\mu_{C1} = \frac{14 + 17 + 28 + 4}{4} = 15.75$$

$$\sigma_{C1} = \sqrt{\frac{(14 - 15.75)^2 + (17 - 15.75)^2 + (28 - 15.75)^2 + (4 - 15.75)^2}{4}}$$

$= 8.555$

Therefore,

$$P(x_{Age} = 25|C_1)\frac{1}{\sqrt{2\pi} * 8.555} * e^{-\frac{(25-15.75)^2}{23.1875}}$$

$= 0.026$

Similarly, $P(x_{Age} = 25|C_2) = 0.010$

Therefore,

$P(X|C_1) = 0.25 * 0.026 = 0.065$
$P(X|C_2) = 0.1667 * 0.01 = 0.0017$

Step 2: Find the class with maximum posterior probability

Now that we have calculated all the probabilities, plug into the formula (eq(12)) and classify the tuple into the class with maximum posterior class probability.

For our example,

$P(C_1|X) = 0.065 * 0.25 = 0.01625$

$P(C_2|X) = 0.0017 * 0.1667 = 0.00028$

Hence, the test data point is a male.

2.3 Pseudo-Code for Naive Bayesian Classifier

```
Begin
class_probability[C_1,...,C_n] = [1, ..., 1]
for C_i in unique(data[Target attribute]){
    class_probablity[C_i] += 1
}
while i < len(class_probablity){
    class_probability[i] = class_probability[i] / len(data + len(class_probablity)
    i ++
}


for x_t in test dataset{
    for C_j in unique(data[Targer Attribute)]{
        Post_prob = class_prob
        for x_i in x_t{
            if(x_i is categorical){
                p = Number of tuples in class j with data[i] = x_i / Total number of tuples in class_j
            }
            else{
                μ = data[i].mean()
                σ = data[i].sigma()
                p = 1/(√(2π)σ) e^- (x-μ_i)^2 / 2σ^2
            }


            Post_prob[j] *= p
        }
        class of x_t = index(max(Post_prob))
     }

}
```

2.4 Additional Notes

- Dealing with 0 - probablity

  This situation may arise if our training dataset doesn't have a categorical value, that our test dataset possess. Plugging this zero value into eq(14) would return a zero probability for P(X—C_i), even though, without the zero probability, we may have ended up with a high probability, suggesting that X belonged to class C_i! A zero probability cancels the effects of all the other (posteriori) probabilities (on C_i) involved in the product.

  There is a simple trick to avoid this problem. We can assume that our training database, D, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the Laplacian correction or Laplace estimator. If we have, say, q counts

to which we each add one, then we must remember to add q to the corresponding denominator used in the probability calculation.

For example,

Suppose that for the class buys computer = yes in some training database, D, containing 1000 tuples, we have 0 tuples with income = low, 990 tuples with income = medium, and 10 tuples with income = high. The probabilities of these events, without the Laplacian correction, are 0, 0.990 (from 990/1000), and 0.010 (from 10/1000), respectively. Using the Laplacian correction for the three quantities, we pretend that we have 1 more tuple for each income-value pair. In this way, we instead obtain the following probabilities (rounded up to three decimal places):

$$\frac{1}{1003} = 0.01; \frac{991}{1003} = 0.988; \frac{11}{1003} = 0.011 \tag{18}$$

respectively. The "corrected" probability estimates are close to their "uncorrected" counterparts, yet the zero probability value is avoided.

- Various empirical studies of this classifier in comparison to decision tree and neural network classifiers have found it to be comparable in some domains. In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers. However, in practice this is not always the case, owing to inaccuracies in the assumptions made for its use, such as class-conditional independence, and the lack of available probability data. Bayesian classifiers are also useful in that they provide a theoretical justification for other classifiers that do not explicitly use Bayes' theorem.

  For example, under certain assumptions, it can be shown that many neural network and curve-fitting algorithms output the maximum posteriori hypothesis, as does the naıve Bayesian classifier.

3. Neural Network Classifier (Back Propagation Network(BPN))

3.1 Introduction

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred to as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be 1 and 1.

In general terms, a neural network is a set of connected input/output units in which each connection has a weight associated with it. The weights are adjusted during the learning phase to help the network predict the correct class label of the input tuples.

These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

Some terms before starting BPN

- **Layer**

  Layer is a general term that applies to a collection of 'nodes' operating together at a specific depth within a neural network.

  There are three type of layer

  - The input layer is contains your raw data.
  - The hidden layer(s) are where the black magic happens in neural networks.
  - And a output layer which gives output.

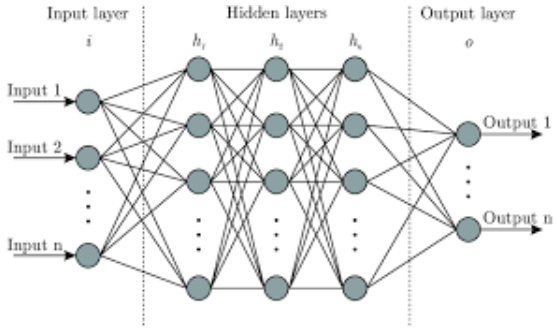


Figure 21: figure explaining different types of a layer in neural network

- **Activation function**
  In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard integrated circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron in neural networks. However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes.

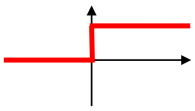
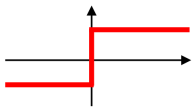
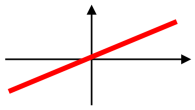
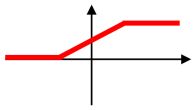
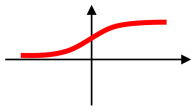
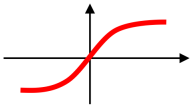
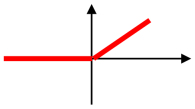
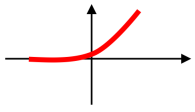
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Figure 22: Some commonly used activation function

- **Weights**
  Weights(Parameters) — A weight represent the strength of the connection between units. If the weight from node 1 to node 2 has greater magnitude, it means that neuron 1 has greater influence over neuron 2. A weight brings down the importance of the input value.

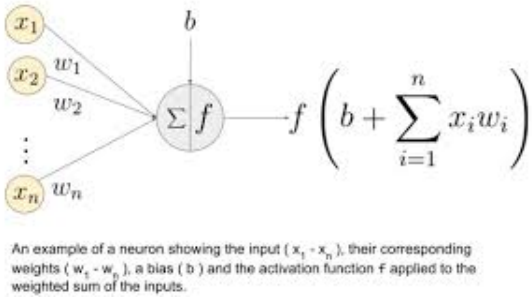


Figure 23: weights and the formula to calculate the value of a node.

- **Learning Rate**
  The learning rate is a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the meansquared distance between the network's class prediction and the known target value of the tuples. The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to 1/t, where t is the number of iterations through the training set so far.

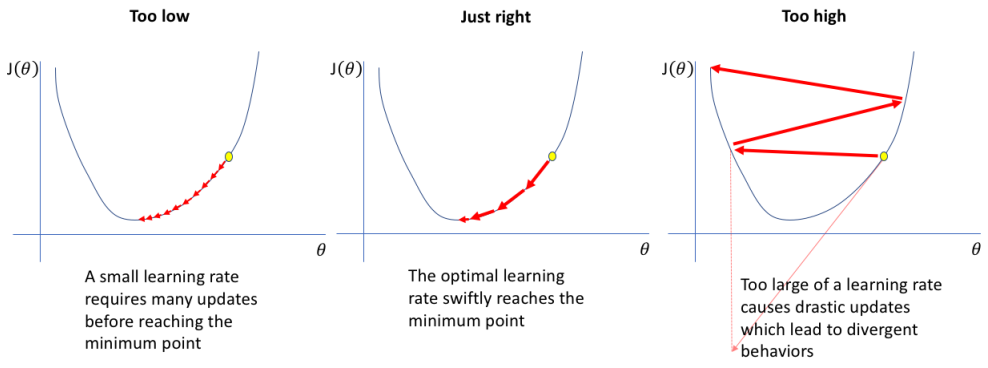


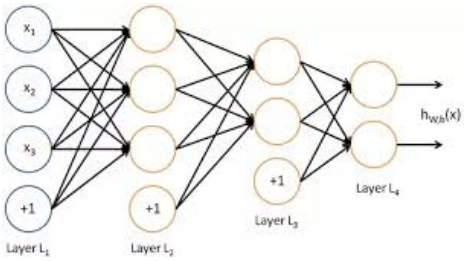
Figure 24: Effects of different Learning rate on convergence.

- **Bias**
  The bias neuron is a special neuron added to each layer in the neural network, which simply stores the value of 1. This makes it possible to move or "translate" the activation function left or right on the graph.

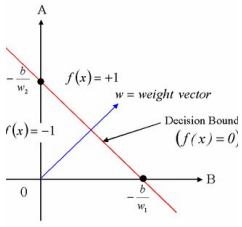
  Without a bias neuron, each neuron takes the input and multiplies it by a weight, with nothing else added to the equation. So, for example, it is not possible to input a value of 0 and output 2. In many cases, it is necessary to move the entire activation function to the left or right to generate the required output values—this is made possible by the bias.

  Although neural networks can work without bias neurons, in reality, they are almost always added, and their weights are estimated as part of the overall model.

  Note that, typically all layer (except the output layer) will have a bias neuron.



(i) Subfigure 1



(ii) Subfigure 2

Figure 25: Subfigure 1 shows a neural network model with with bias.
Subfigure 2 shows the effect of bias on the decision boundary.

- **Architecture**
  The number of layers and node in each layer constitutes the architecture of a neural network.

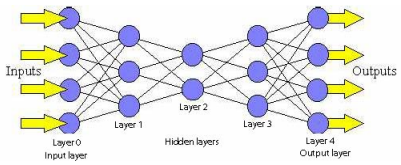


Figure 26: Illustration of architecture

- **Epoch**
  An epoch is a measure of the number of times all of the training vectors are used once to update the weights.

  For batch training all of the training samples pass through the learning algorithm simultaneously in one epoch before weights are updated.

3.2 Back Propagation Neural Network

Back-propagation is the essence of neural net training. It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

Backpropagation is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps to calculate the gradient of a loss function with respects to all the weights in the network.

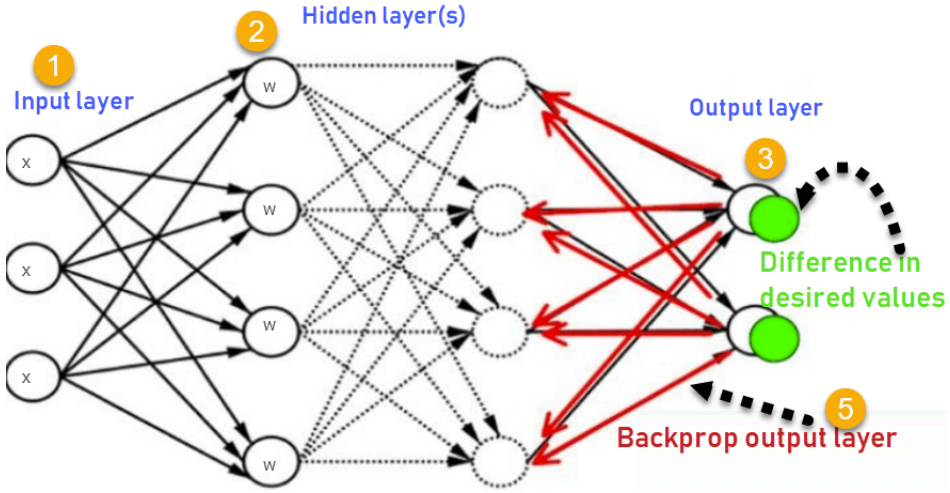


Figure 27: Simple working of BPN

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program t has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

Types of BPNs

Two Types of Backpropagation Networks are:

- **Static back-propagation**
  It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.
- **Recurrent Backpropagation**
  Recurrent backpropagation is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.

The main difference between both of these methods is that the mapping is rapid in static back-propagation while it is non-static in recurrent backpropagation.

3.3 Classifying using BPN with example

Example Dataset

Table 2: Example dataset

class	x	y
class 1 (w1)	2	2
class 1 (w1)	-1	2
class 1 (w1)	1	3
class 1 (w1)	-1	-1
class 1 (w1)	0.5	0.5
class 2 (w2)	-1	-3
class 2 (w2)	0	-1
class 2 (w2)	1	-2
class 2 (w2)	-1	-2
class 2 (w2)	0	-2

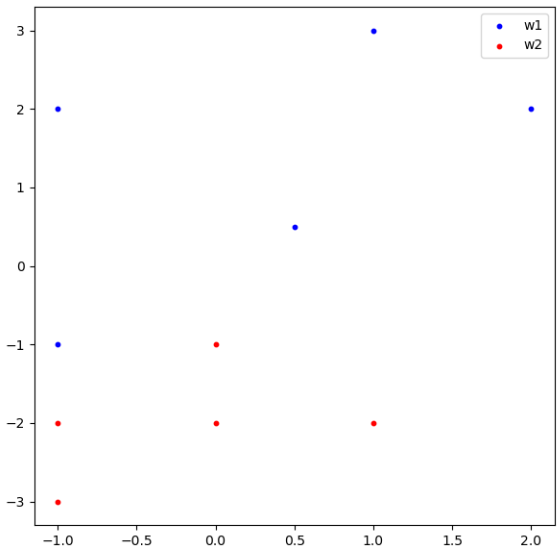


Figure 28: Visualisation of the dataset

Step 1: Determine the architecture of the neural network

The architecture of the neural network will depend on the problem. For linearly separable problem, having one hidden layer is enough. For non linearly separable and more complex problem, having more hidden layers would be better.

The number of nodes in the input layer would be the dimension of the data points + 1 (for bias). There is no constraint on the number of nodes in the input layer, but its better to have them equal to or greater than the nodes in the input layer. The number of nodes in the output layer would be the number of distinct class for the classification problem.

For this example, we are going with a neural network with 3 layer (1 - input, 1 - hidden, 1 - output) with 2 + 1 (bias) nodes for input, 2 + 1 (bias) nodes for hidden and 2 nodes for output.

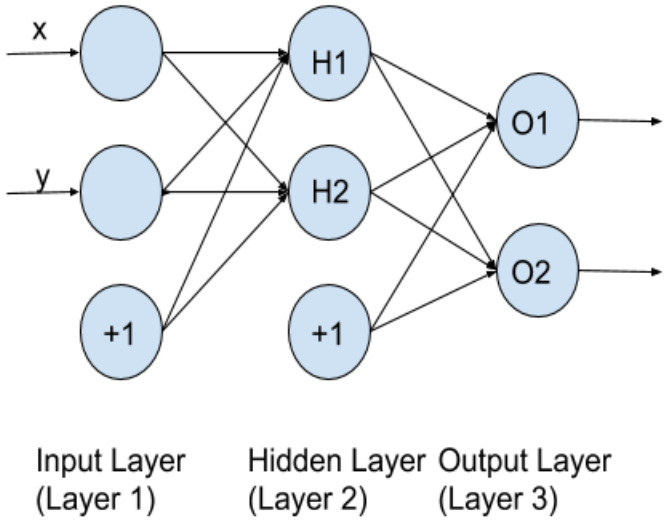


Figure 29: Architecture of the Neural Network

Step 2: Deciding the weights, learning rate, activation function and number of epoch

We need to assign value to weights, learning rate, activation function and tha max epoch value. The value of learning rate is depends upon the problem we are solving. The best way to assign weights is to randomly assign values between 0 and 1.

Here, we are using learning rate to be 0.5, running the neural network for 1000 epoch and sigmoid function as activation function.(problem dependent)

A little notation,

W^i_{jk} means the weight that is present at the j^{th} node of i^{th} layer to the k^{th} node of $(i+1)^{th}$ layer.

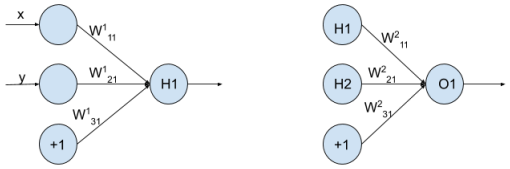


Figure 30: Illustration of the above notation

Table 3: Value of weights

Weights	Value
W^1_{11}	0.13436424411240122
W^1_{21}	0.8474337369372327
W^1_{31}	0.763774618976614
W^1_{12}	0.2550690257394217
W^1_{22}	0.49543508709194095
W^1_{32}	0.4494910647887381
W^2_{11}	0.651592972722763
W^2_{21}	0.7887233511355132
W^2_{31}	0.0938595867742349
W^2_{12}	0.02834747652200631
W^2_{22}	0.8357651039198697
W^2_{32}	0.43276706790505337

Step 3: Forward propagation

Let Out_j be the output of j^th node of a layer.
Let W_{ij} denote the weight connecting i_{th} node of a layer to the j_{th} node of the next layer.
These conventions make it easier to talk about two successive layers.
We need to forward propagate the neural network to get the ouptput value given by our network.

Forward is nothing but the calculation of output nodes for the input vector.

Initially the output would be erroneous and we need to propagate this error to update weights at each level/ layer.

Value at a node is given by,

$$N_k^{i+1} = f(\sum_{j=1}^{n} W_{jk}^i * N_j^i) \tag{19}$$

where,

- N_y^x is the x^{th} node in y^{th} layer (it also includes the bias node).
- n is the total number of nodes in i^{th} layer.
- f(.) is the activation function.

Therefore, if we use the above formula, we get, N_1^1 (H_1) = f($W^1$11 * x + $W^1$21 * y + $W^1$31 * 1)
N_2^1 (H_2) = f($W^1$12 * x + $W^1$22 * y + $W^1$32 * 1)
N_1^2 (O_1) = f($W^2$11 * H_1 + $W^2$21 * H_2 + $W^2$31 * 1)
N_2^2 (O_2) = f($W^2$12 * H_1 + $W^2$22 * H_2 + $W^2$32 * 1)

When we plug in the values for the first value of the dataset i.e. (2, 2), We get,

H_1 = f(0.13436424411240122 * 2 + 0.8474337369372327 * 2 +
0.763774618976614 * 1)
=f(2.7273705810758817)
= 0.9386225302230806

H_2 = f(0.2550690257394217 * 2 + 0.49543508709194095 * 2 +
0.4494910647887381 * 1)
= f(1.9504992904514635)
= 0.8755010741482664

O_1 = f(0.651592972722763 * 0.9386225302230806 + 0.7887233511355132 * 0.8755010741482664
+ 0.0938595867742349 * 1)
= f(1.3959875726318156)
= 0.8015464048450159

O_2 = f(0.02834747652200631 * 0.9386225302230806 + 0.8357651039198697 * 0.8755010741482664 + 0.43276706790505337 * 1)
= f(1.1910878942610617)
= 0.7669355767878618

Step 4: Calculating the Error and Back Propagating it

The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit j in the output layer, the error Err_j is computed by,

$$Err_j = Out_j(1 - Out_j) * (T_j - Out_j) \tag{20}$$

where,

- Out_j(1 - Out_j) is the derivative of the logistic (sigmoid) function.
- T_j is the target value of the O_j node.

To compute the error of a hidden layer unit j, the weighted sum of the errors of the units connected to unit j in the next layer are considered. The error of a hidden layer unit j is,

$$Err_j = Out_j(1 - Out_j) * \sum_k Err_k * W_{jk} \tag{21}$$

where,

- O_j(1 - O_j) is the derivative of the logistic (sigmoid) function.
- W_{jk} is the weight of the connection from unit j to a unit k in the next higher layer.
- Err_k is the is the error of unit k.

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δ W_{ij} is the change in weight W_{ij}:

$$\Delta W_{ij} = (l) * Err_j * Out_i \tag{22}$$
$$W_{ij} = W_{ij} + \Delta W_{ij} \tag{23}$$

where,

- l is the learning rate.

In our example, the error for Err_1 for the output layer is,
Err_i = 0.8015464048450159 * (1 - 0.8015464048450159) * (1 - 0.8015464048450159)
= 0.03156796688859639

The change in weight for Hidden Layer is,
$\Delta\ W_{11}$ = (0.5) * (0.03156796688859639) * (0.9386225302230806)
= 0.014815202477486385

W_{11} = 0.651592972722763 + 0.014815202477486385
= 0.6664081752002493

Similarly, the error for Err_1 for the Hidden Layer is,
Err_i = 0.9386225302230806 * (1 - 0.9386225302230806) *
(0.03156796688859639 * 0.651592972722763 +
(0.7669355767878618) * (1 - 0.7669355767878618) * (0 - 0.7669355767878618) * 0.02834747652200631)
= 0.0009611362816286504

The change in weight for Hidden Layer is,
$\Delta\ W_{11}$ = (0.5) * (0.0009611362816286504) * (2)
= 0.0009611362816286504

W_{11} = 0.13436424411240122 + 0.0009611362816286504
= 0.13532538039402986

By doing the same for all weights, we get,

Table 4: Updated value of weights

Weights	Updated Value
W_{11}^1	0.13532538039402986
W_{21}^1	0.8483948732188613
W_{31}^1	0.7642551871174283
W_{12}^1	0.24529471187779883
W_{22}^1	0.4856607732303181
W_{32}^1	0.4446039078579267
W_{11}^2	0.6664081752002493
W_{21}^2	0.8025422455953347
W_{31}^2	0.10964357021853309
W_{12}^2	-0.03598862367938312
W_{22}^2	0.7757555441456759
W_{32}^2	0.3642239655078642

Step 5: Forward Propagate and Repeat

We now, just forward propagated and back propagated for one data-point. We need to do that for all datapoints in our data set, and we need to do these max_epoch number of times.

We have a max_epoch, as it is unlikely we get all errors as zero.

Step 6: Output Visualisation (Additional step)

Here are the weights and decision boundary for our neural network after 1000 epoch.

Table 5: Updated value of weights

Weights	Updated Value
W_{11}^1	-4.165736617684192
W_{21}^1	4.494165872980581
W_{31}^1	2.064357120666933
W_{12}^1	-2.924097575618484
W_{22}^1	3.3401443204537666
W_{32}^1	1.4433168050534642
W_{11}^2	5.429646386040146
W_{21}^2	4.037845396293455
W_{31}^2	-4.237515602055247
W_{12}^2	-5.892609497304456
W_{22}^2	-3.4770990687653325,
W_{32}^2	4.209278121692298

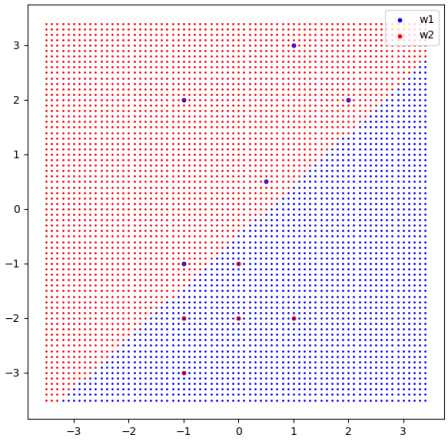


Figure 31: Classification using our neural network

3.4 Pseudo-Code for BPN

```
Begin
Initialize all weights and biases in network;
while terminating condition is not satisfied {
 for each training tuple X in D {
 // Propagate the inputs forward:
     for each input layer unit j {
         O_j = I_j; // output of an input unit is its actual input value
         for each hidden or output layer unit j {
         I_j = sum_i w_ij O_i + theta_j; //compute the net input of unit j with respect to
    the previous layer, i
         O_j = 1 / (1 + e^{-I_j}) ; // compute the output of each unit j
     // Backpropagate the errors:
    for each unit j in the output layer
         Err_j = O_j(1 - O_j)(T_j - O_j); // compute the error
    for each unit j in the hidden layers, from the last to the first hidden
    layer
         Err_j = O_j(1 - O_j) sum_k Err_k * w_jk; // compute the error with respect to the
    next higher layer, k
    for each weight w_ij in network {
         Delta w_ij = (l) * Err_j * O_i;
         // weight increment
         w_ij = w_ij + Delta w_ij; // weight update
    for each bias theta_j in network {
         Delta theta_j = (l) * Err_j; // bias increment
         theta_j = theta_j + Delta theta_j;
         } // bias update
 } }
```

3.5 Additional Notes

Disadvantages

- Knowledge Representation

  A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

  Various algorithms for rule extraction have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

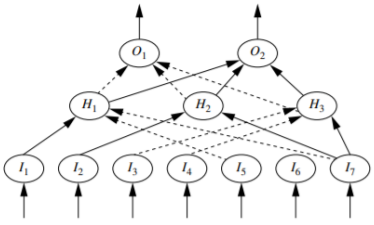
  Fully connected networks are difficult to articulate. Hence, often the first step in extracting rules from neural networks is network pruning. This consists of simplifying the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal does not result in a decrease in the classification accuracy of the network.

  Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used

to find the set of common activation values for each hidden unit in a given trained twolayer neural network (Figure 9.6). The combinations of these activation values for each hidden unit are analyzed. Rules are derived relating combinations of activation values with corresponding output unit values. Similarly, the sets of input values and activation

values are studied to derive rules describing the relationship between the input layer and the hidden "layer units"? Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including M-of-N rules (where M out of a given N conditions in the rule antecedent must be true for the rule consequent to be applied), decision trees with M-of-N tests, fuzzy rules, and finite automata.

Sensitivity analysis is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this analysis form can be represented in rules such as "IF X decreases 5



Identify sets of common activation values for each hidden node, H_i:
for H_1: (–1,0,1)
for H_2: (0,1)
for H_3: (–1,0.24,1)

Derive rules relating common activation values with output nodes, O_j:
IF (H_2=0 AND H_3=–1) OR
(H_1=–1 AND H_2=1 AND H_3=–1) OR
(H_1=–1 AND H_2=0 AND H_3=0.24)
THEN O_1=1, O_2=0
ELSE O_1=0, O_2=1

Derive rules relating input nodes, I_j, to output nodes, O_j:
IF (I_2=0 AND I_7=0) THEN H_2=0
IF (I_4=1 AND I_6=1) THEN H_3=–1
IF (I_5=0) THEN H_3=–1

Obtain rules relating inputs and output classes:
IF (I_2=0 AND I_7=0 AND I_4=1 AND I_6=1) THEN class=1
IF (I_2=0 AND I_7=0 AND I_5=0) THEN class=1

Figure 32: Knowledge representation of a neural network.

- The actual performance of backpropagation on a specific problem is dependent on the input data.
- Backpropagation can be quite sensitive to noisy data
- You need to use the matrix-based approach for backpropagation instead of mini-batch.

4. Support Vector Machine (SVM)

4.1 Introduction

SVM is a method for the classification of both linear and nonlinear data. In a nutshell, an SVM is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

Intuition and Math behind SVM

For linearly separable data, a hyper plane divides the hyperspace in such a way that, points to one side of hyperplane is class 1 and the points on the other side of the hyperplane is class 2.

In mathematical sense, if

$$w^t x + b = 0 \tag{24}$$

is the equation of the hyperplane, then, $w^t x$ + b ¿ 0, all x satisfying this equation, will be of one class and the points satisfying the eq
$w^t x$ + b ¡ 0
will be another class.

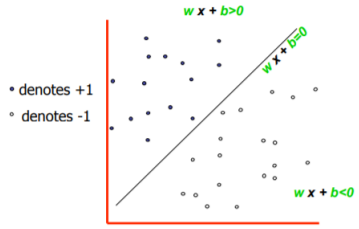


Figure 33: Illustration of how the hyperplane separates points.

We can easily separate the datapoints with many hyperplanes, For example,

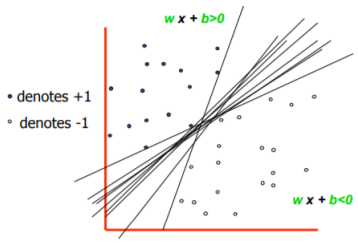


Figure 34: This problem has infinitely many solutions.

But only one of those hyerplane classify the data optimally, and we use svm to idenitify them.

The optimal plane is the one in which the distance between the plane and the class points are maximum. Moreover, the minimum distance between points in class A and the hyperplane and points in class B and the hyperplane must be same, i.e. if the minimum distance of points class A with the hyperplane is greater than that of minimum distance of points in classe B with the hyper plane, then the hyperplane wouldnt be optimum as we tend to classify points more towards class A than class B. For example, consider this one dimensional data set,

Table 6: Example dataset

class	x
class A	1
class A	2
class A	3
class B	5
class B	6
class B	7

The optimum plane (or point) in this case would be x = 4, which is equally spaced between both the classes.

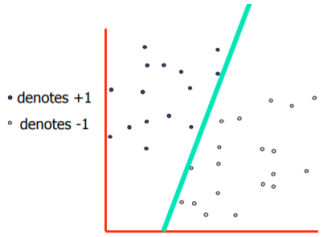


Figure 35: Optimal hyper plane

Support vectors are those planes which run parallel to the decision boundary and pass through the points of both the classes, which have minimum distance with the hyperplane.

The equation of the support vectors is,

$$w^t x + b = \pm 1 \tag{25}$$

The optimal decision boundary is the one in which the margin (distance between the support vectors) is maximum.

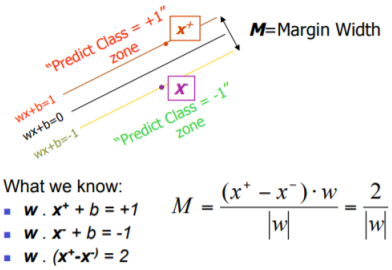


Figure 36: Optimal hyper conditions

From above figure, we can tell that,

$$w^t x + b \geqslant 1 if x \in classA \tag{26}$$
$$w^t x + b \geqslant -1 if x \in classB \tag{27}$$

We can create a label such that, points in class B have label -1 and point in class A has label 1.

Therefore, the above equation becomes,

$$w^t x_i + b \geqslant 1 if y_i \in classA \tag{28}$$
$$w^t x_i + b \geqslant -1 if y_i \in classB \tag{29}$$

We can combine them into one equation,

$$y_i(w^t x_i + b) \geqslant 1 \tag{30}$$

Moreover, we know that margin width,

$$M = \frac{2}{|w|} \tag{31}$$

Maximising M, is same as minimising —w—.

Thus, we minimise,

$$|w| \tag{32}$$

$$and \forall (y_i, x_i) : y_i(w^t x_i + b \geqslant 1) \tag{33}$$

This minimisation problem can be solved by associating Lagrange multiplier with each constraint of this problem.

Cases of SVM

There are three cases of SVM. They are:

- **Hard Margin**

  Here, we try to find a decision boundary that classifies all training points correctly.

  Extremely useful in the case of linearly seperable, but performs poorly in case of non linearly separable and is susceptible to noises.

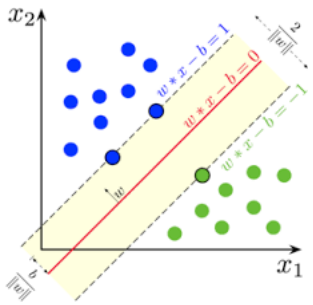


Figure 37: Example of Hard Margin.

- **Soft Margin**

  Here, we try to find a decision boundary that classifies most of the points correctly. We allow some points to be misclassified. The amount of error allowed is set by us.

  This is useful to classify noisy data and some non-linearly separable data.

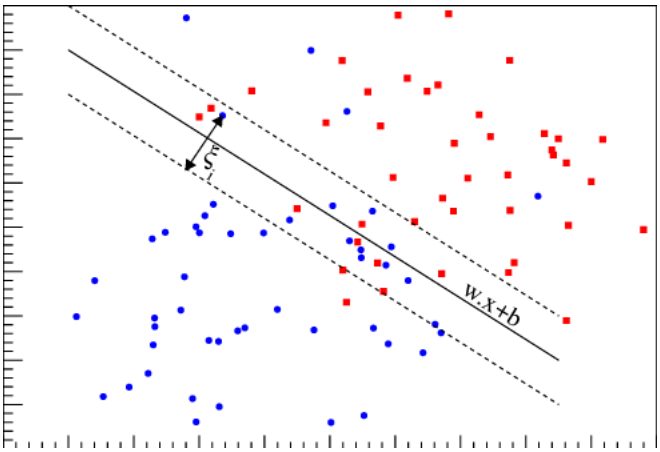


Figure 38: Example of Soft Margin.

- **Kernal**
  Here, we transform the points to a higher dimension (where they are linearly separable) and find a linear hyperplane that classifies the points accurately in that dimension and we plot the contours of that hyperplane on our lower dimension to get a decision boundary.

  This is extremely useful to classify non-linearly seperable data.

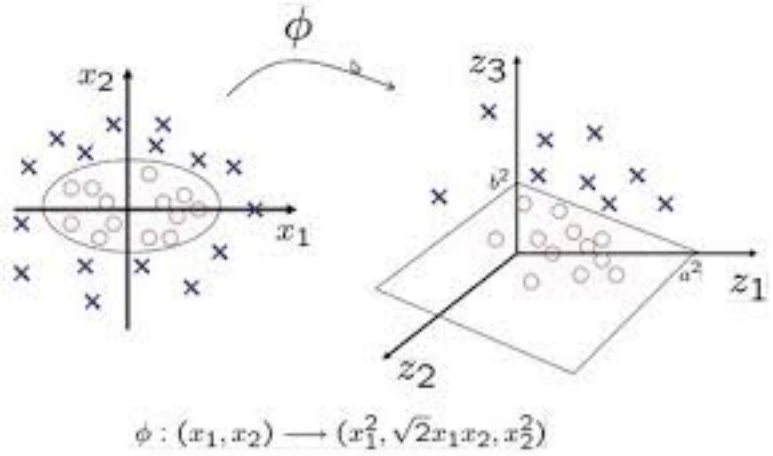


Figure 39: Example of the use of kernal function to transform the input to higher dimension, where they are linearly separable.

4.2 Classifying using SVM with example

Example Dataset

Table 7: Example dataset

class	x	y
class 1 (w1)	2	2
class 1 (w1)	-1	2
class 1 (w1)	1	3
class 1 (w1)	-1	-1
class 1 (w1)	0.5	0.5
class 2 (w2)	-1	-3
class 2 (w2)	0	-1
class 2 (w2)	1	-2
class 2 (w2)	-1	-2
class 2 (w2)	0	-2

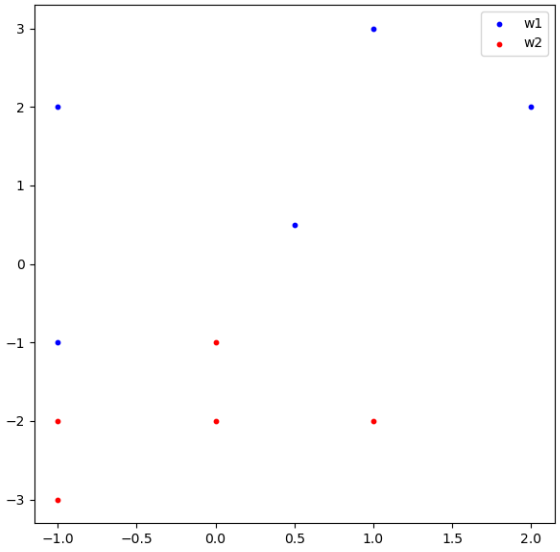


Figure 40: Visualisation of the dataset

Case 1 : Hard Margin

Step 1: Decide the initial weights of the plane.

The Equation of the plane that separates the classes is :

$$w^t x + b = 0 \tag{34}$$

where,

- w is the normal vector of the hyperplane that separates the data. It is represented as a matrix. Hence w^t means transpose of w matrix.
- x is a variable vector.
- b is the bias.

The z matrix would be a matrix where, $z_i = 1$ if x_i in class 1
z_i = -1 if x_i in class 2

Step 2: Finding the Lagrange Multipliers

We need to find the lagrange multipliers for our hyperplane, which inturn would help us to identify the equation of the hyperplane.

$$L(w, b, \alpha) = \frac{||w||^2}{2} - \sum_{k=1}^{n} \alpha_k (1 - z_k(w^t x_k + b)) \tag{35}$$

where,

- w^t is the transpose of weight matrix, and $||w||$ is the magnitude of the normal vector of the hyper plane.
- b is the bias.
- α is a matrix of all the lagrange multipliers. There are totally n lagrange multiplier, where n is the number of datapoints.
- z_k is the class value of a data point x_k. Note that z_k takes either 1 or -1 and not 1 or 0.

After finding few relation between α and weight matrix and bias, the above equation reduces to,

$$L(\alpha) = \sum_{k=1}^{n} \alpha_k - \frac{1}{2} \sum_{k,j}^{n} \alpha_k \alpha_j z_k z_j x_j^t x_k \tag{36}$$

We should, find the minimum value of L(α), which satisfy the following constraint,

(a) $\sum_{k=1}^{n} z_k \alpha_k = 0$
(b) $\alpha_k \geqslant 0, k = 1, 2, ..., n$

For our example, we got,

Step 3: Finding Weight Vector and Bias

The weight vector is given by,

$$w = \sum_{k=1}^{n} \alpha_i z_i x_i \tag{37}$$

We know that,
$z_i(w^t x_i + b) = 1$
is the equation of the support planes.

And these support planes would pass through the corner points(that are to the other class end) of each class.

Table 8: *αvalues*

α	Value
α_1	1.26183502 x 10^{-1}
α_2	1.03087027 x 10^{-14}
α_3	2.58278642 x 10^{-15}
α_4	3.45312250 x 10^{0}
α_5	4.21384434 x 10^{1}
α_6	1.64516838 x 10^{-15}
α_7	3.01092476 x 10^{0}
α_8	1.82009356 x 10^{-14}
α_9	9.89765678 x 10^{1}
α_10	8.76371458 x 10^{-15}

Therefore, we need to find the points.

α_i would be negligible for all non - other points/ points through which the support vector wont pass through.

Therefore, with this we can locate the corner point and take avgerage to find avgerage b value.

In our example we get,

w = [-2.0002976 2.00039283] and b = 1.0001476102615467.

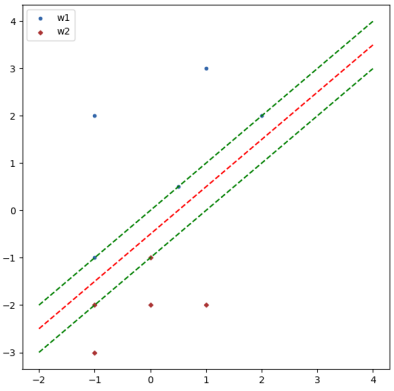


Figure 41: The decision boundary for the example dataset. The red line is the actual decision boundary with green line as the support vectors.

Case 2 : Soft Margin

The steps are same as that of Hard Margin, we only change the equation to allow error.

$$w^t x + b = 1 - \varepsilon \tag{38}$$

$$L(w, b, \varepsilon, \alpha, \gamma) = \frac{||w||^2}{2} + C * \sum_{k=1}^{n} \varepsilon_k - \sum_{k=1}^{n} \alpha_k (z_k (w^t x_k + b) - 1 + \varepsilon_k) - \sum_{i=k}^{n} \gamma_k \varepsilon_k \tag{39}$$

where,

- ε is the error in our classifier.
- $\alpha and \gamma$ are two lagrange multiplier.
- C is a regularisation parameter, which tells us how much error to be present. If C = 0, we get hard margin, and if C is a large value, then we allow a lot of miscalculation as a the error term in eq(39) increases and dominates the equation.

This may seem difficult to solve, but they give the same equation,but differ in constraint.

$$L(\alpha) = \sum_{k=1}^{n} \alpha_k - \frac{1}{2} \sum_{k,j}^{n} \alpha_k \alpha_j z_k z_j x_j^t x_k \tag{40}$$

We should, find the minimum value of L(α), which satisfy the following constraint,

(a) $\sum_{k=1}^{n} z_k \alpha_k = 0$
(b) $C \geqslant \alpha_k \geqslant 0, k = 1, 2, ..., n$

The weight vector is given by,

$$w = \sum_{k=1}^{n} \alpha_i z_i x_i \tag{41}$$

The error or slack is giver by,

$$\varepsilon_i = \frac{\alpha_i}{C} \tag{42}$$

Similarly, corresponding α of non corner point will be negligible. With this we find the corner points.

Same way we substitute the corner points in eq(40) to find bias .

Case 3 : Kernal method

Here we transform the input data to a higher dimension using kernal functions.

$$y = \Phi(x) \tag{43}$$

where,

- y is the transformed data.
- x is the input data points.
- $\phi(.)$ is a transformation or kernal function.

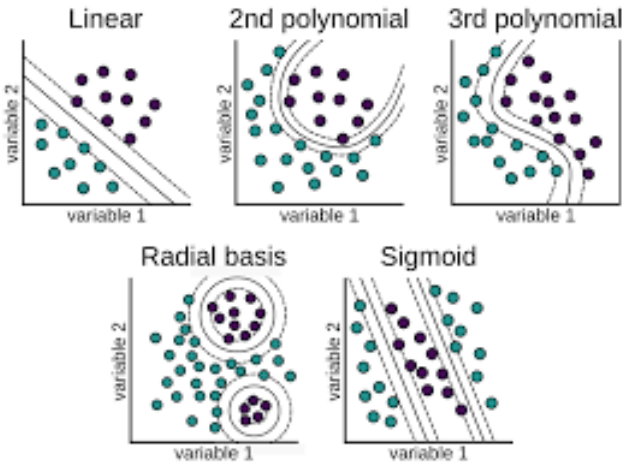


Figure 42: Different kernal functions

After transforming the input data, follow hard margin algorithm to get the equation of plane in the higher dimension.

To make the computation at higher dimension easier, we use a special property of kernal functions. i.e. the dot product of the function is conserved throughout the dimensions.

The contours of that plane on the input dimension would be our decision tree.

4.3 Pseudo-Code SVM

Hard Margin

```
Begin
Initialize all weights to 0;
for each training tuple X in D {
z_i = classlabel(X)
}
n = len(X)
L(α) = Σ_{k=1}^{n} α_k − (1/2) Σ_{k,j}^{n} α_k α_j z_k z_j x_j^t x_k
α = α : L(α) is maximum,
        α_i ⩾ 0, and
        Σ_{i=1}^{n} α_i z_i = 0

Weights = Σ_{i=1}^{n} z_i α_i x_i

for i in len(α){
if(alpha_i ⩾ 0.001)
  b = \dfrac{1}{z_i} − w^t x_i
  add b to the set bias
}
bias = bias.mean()
```

Soft Margin SVM

```
Begin
Initialize all weights to 0;
for each training tuple x in X {
z_i = classlabel(x)
}
initialise the desired value of c.
n = len(X)
L(α) = Σ_{k=1}^{n} α_k − (1/2) Σ_{k,j}^{n} α_k α_j z_k z_j x_j^t x_k
α = α : L(α) is maximum,
        c ⩾ alpha_i ⩾ 0, and
        Σ_{i=1}^{n} α_i z_i = 0

Weights = Σ_{i=1}^{n} z_i α_i x_i
for i in len(X){
 ε_i = \dfrac{α_i}{c}
}
for i in len(X){
if(alpha_i ⩾ 0.001)
  b = \dfrac{1−ε_i}{z_i} − w^t x_i
  add b to the set bias
}
bias = bias.mean()
```

Kernal trick SVM

```
Begin
Initialize all weights to 0;
for each training tuple x in X {
z_i = classlabel(x)
}
Y = ψ(X) //transforming the input into higher dimensions.

weight, bias = HardMargin(Y)
```

4.4 Additional Notes

More about Kernal functions

Consider the following example. A 3-D input vector X = (x1, x2, x3) is mapped into a 6-D space, Z, using the mappings ϕ_1(X) = x1, ϕ_2(X) = x2, ϕ_3(X) = x3, ϕ_4(X) = $(x1)^2$, ϕ_5(X) = x1.x2, and ϕ_6(X) = x1.x3. A decision hyperplane in the new space is

$$d(Z) = W^t Z + b \tag{44}$$

where, W and Z are vectors. This is linear. We solve for W and b and then substitute back so that the linear decision hyperplane in the new (Z) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to

Given the test tuple, we have to compute its dot product with every one of the support vectors. In training, we have to compute a similar dot product several times in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly.

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, ϕ(Xi)· ϕ(Xj), where ϕ(X) is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a kernel function, K(Xi, Xj), to the original input data. That is,

$$K(Xi, Xj) = \phi(Xi).\phi(Xj) \tag{45}$$

In other words, everywhere that ϕ(Xi)· ϕ(Xj) appears in the training algorithm, we can replace it with K(Xi ,Xj). In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane. kernel functions that could be used to replace the dot product scenario just described,

$$Polynomialkernelofdegreeh : K(Xi, Xj) = (Xi\Delta Xj + 1)^h \tag{46}$$

$$Gaussianradialbasisfunctionkernel : K(Xi, Xj) = e^{-\frac{||kXiXjk||^2}{2\sigma^2}} \tag{47}$$

$$Sigmoidkernel : K(Xi, Xj) = tanh(\kappa XiXj - \delta) \tag{48}$$

5. Genetic Algorithm

5.1 Introduction

Genetic Algorithm are search algorithms based on the mechanism of natural selection and nature genetics. They combine survival of the fittest among the string structure with randomized information exchange to form a search algo with some innovative flair of human search. In every generation, a new set of artificial creatures (string), is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure.

Genetic algorithm has three operators,

- **Selection**

  The idea is to give preference to the individuals with good fitness scores and allow them to pass there genes to the successive generations.

- **Crossover**

  This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).

- **Mutation**

  The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence.

Some terms before starting GA

- **Fitness function**

  This is the function we are trying to optimise in an interval. Each people in the generation would be evaluated based on this metric.

- **Chromosome**

  Each string is called chromosome.

- **Generation**

  Generation is a collection of chromosomes.

- **Gene**

  Each bit in the chromosome is called gene.

Here is a flowchart for Genetic Algorithm,

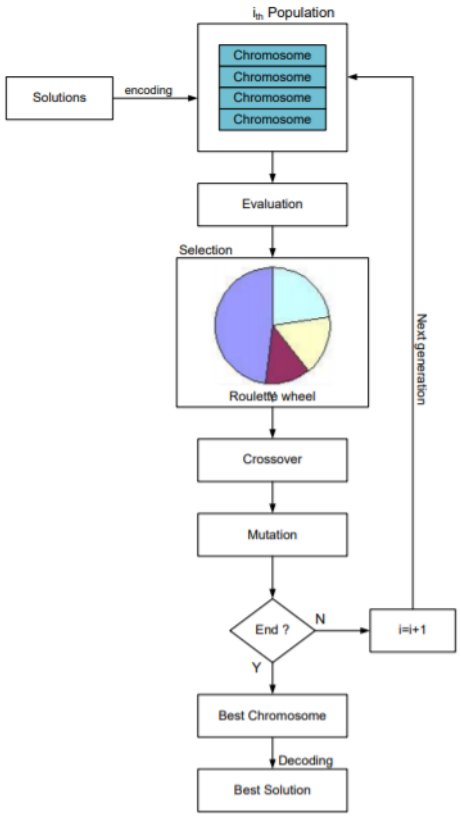


Figure 43: Flow chart for Genetic Algorithm

5.2 Optimising a function using Genetic Algorithm with example

Maximise the following function with the constraint on x. Function to be optimised is :

$$f(x) = x^3 - 2x^2 + x \tag{49}$$

$$\tag{50}$$

where, x $\in [1, 31]$

Step 1: Decide upon an encoding

In GA, we tend to work with strings. Therefore, we need to convert our input domain into string.

In our example, we take the binary representation of x as the string encoded x, i.e. x = 5 will be viewed as '00101'. Moreover, the max vale of x is 31, and min value is 1, 5-bit representation is enough to represent all values of x.

Step 2: Randomly Populate.

In this step, we initially decide on the population size and then we randomly populate the generation, i.e. randomly choose value of x between [1, 31] and add it to initial generation.

In our example, we chose 4 as population size per generation.

The initial generation is :

Table 9: GEN 1

value
11010 (26)
10101 (21)
11001 (25)
00001 (1)

Step 3: Find Fitness value, percentage and expected count for each chromosome in the generation

We need to calculate the fitness sores and percentage. Fitness percentage and expected count is given by,

$$fit_perc(chromosome) = \frac{fitness(chromosome)}{\sum_{g \in chromosome} fitness(g)} \tag{51}$$

$$expected_count = fit_perc(chromosome) * population_size \tag{52}$$

In our example, we get fitness score and percentage as,

Table 10: Fitness calculation

value	fitness score	fitness_precentage	expected_count
11010 (26)	16250	0.429	1.716
10101 (21)	7220	0.191	0.764
11001 (25)	14400	0.38	1.520
00001 (1)	0	0	0

Clearly, the chromosome 26 is the most fit.

Step 4: Selecting chromosome for Reproduction

Here we randomly choose chromosomes from the current generation for reproduction.

In our example, we are following roulette wheel, where the probability of a chromosome being chosen is its $fitness_percentage$. This way, we give maximum probability for fit pass on its trait to the next generation, and also keep the process random to let weak traits also a chance to reproduce.

We need to choose chromosomes population number of times.

In our example,

Table 11: Russian Roulette

value	fitness score	$\text{fitness}_percentage$	expected_count	Actual Count from Russian Roulette
11010 (26)	16250	0.416	1.664	2
10101 (21)	8400	0.215	0.86	0
11001 (25)	14400	0.369	1.476	2
00001 (1)	0	0	0	0

Step 5: Randomly Reproduce

Now that we have our reproduction candidates ready, we need to randomly mate them / crossover them.

Randomly pair them up and reproduce.

Reproduction is simple, we need to choose a point on the two chromosomes and the first part of one chromosome and the second part of the second chromosome would make a child. Similarly, first part of second chromosome and second part of first chromosome would another child.

In our example, we have mated, 26 and 26 and 25 and 25.

The point randomly chosen for the first reproduction is 2, Therefore, the offsprings are 11 + 010 = 11010 and 11 + 010 = 11010.

Similarly after doing this for the other pair, we get,

The offsprings of this mating are : [26, 26, 25, 25]

Step 6: Mutation

This helps to mutate the children chromosomes. We need to decide upon the rate of mutation. The number of mutation per generation would be,

$$number\ of\ mutation = rate * total\ number\ of\ genes\ in\ the\ generation. \quad (53)$$

And then we randomly choose an integer between [1, total number of genes], number of mutation times and change the values of those genes i.e. consider all the chromosome to be a single string, and change accordingly.

In our example, mutation rate is 0.1, hence only 2 genes would be modified per generation.

randomly chosen position is : [3 and 7]

The total string = 11010110101100111001

Mutated string would be = 11110110101100110001

Therefore, the next generation would be : [30, 26, 25, 17]

Step 7: Keep on moving to the next generation until termination condition is met

The typical termination conditions are,

- When there has been no improvement in the population for X iterations. we reach an absolute number of generations.
- When the objective function value has reached a certain pre-defined value.

In our example, we chose the first one as termination condition, i.e. after 10 continuous generations, if the maximum fitness of the previous generation is $greater_then$ the current maximum fitness, we stop

The final generation was : [31, 24, 31, 24] and terminated after 45 generations.

The solution would the most fit chromosome in this generation.

5.3 Pseudo-Code for GA

```
Begin
Encode(input_domain)
gen = rand(input_domain, population_size)
while(termination condition is not met){
    fitness_value = function_to_be_optimised(gen)
    fitness_percentage = fitness_value / fitness_value.sum()
    expected_count = fitness_value * population_size
    candidates_for_reproduction = rand(gen, probability = fitness_percentage,
    n = population_size)
    mated = []
    new_generation = []
    while(mated != gen){
        parents = rand(gen, 2)
        if(parents not in mated){
            k = rand(len(gen[0]))
            new_generation.append(parents[0][0 : k] + parents[1][k : len(
    parants[0]))
            new_generation.append(parents[1][0 : k] + parents[0][k : len(
    parants[0]))
            mated.append(parents)
    }

    gene_string = ''
    for chromosomes in new_generation{
        gene_string = gene_string + chromosome
    }

    mutated_positions = rand(len(gene_string), approximate(
    mutations_per_generatio * len(gene_string)))

    for pos in mutated_position{
        mutated_value = rand(possible_gene_value)
        while(mutated_value == gene_string[pos])
            mutated_value = rand(possible_gene_value)
        gene_string[pos] = mutated_value
    }
    new_generation = []
    i = 0
    while(len(new_genertion) < len(gen)){
        new_generation.append(gene_string[i : i + len(gen[0]])
        i = i + len(gen[0]]
        }
    gen = new_generation
}
optimal_sol = gene[max(fittness_value).index]
```

6. Bucket Brigade Algorithm (BBC)

6.1 Introduction

Classifier systems in machine learning is useful to distinguish three levels of activity when looking at learning from the point of view of classifier systems: At the lowest level is the performance system. This is the part of the overall system that interacts directly with the environment. It is much like an expert system, though typically less domain-dependent. The performance systems we will be talking about are rule-based, as are most expert systems, but they are message-passing, highly standardized, and highly parallel. Rules of this kind are called classifiers. These Rules must be evaluated for their performance, and here is where we use BBC.

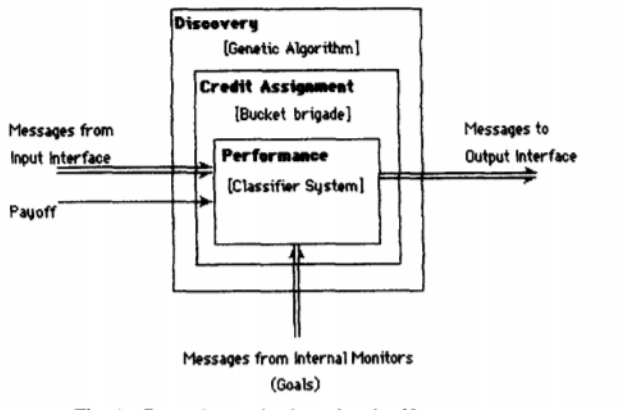


Figure 44: General organisation of Classifier System

The third level of activity, the rule discovery system, is required because, even after the system has effectively evaluated millions of rules, it has tested only a minuscule portion of the plausibly useful rules. Selection of the best of that minuscule portion can give little confidence that the system has exhausted its possibilities for improvement; it is even possible that none of the rules it has examined is very good. The system must be able to generate new rules to replace the least useful rules currently in place. The rules could be generated at random (say by "mutation" operators) or by running through a predetermined enumeration, but such "experience-independent" procedures produce improvements much too slowly to be useful in realistic settings. Somehow the rule discovery procedure must be biased by the system's accumulated experience. In the present context this becomes a matter of using experience to determine useful "building blocks" for rules; then new rules are generated by combining selected building blocks. Under this procedure the new rules are at least plausible in terms of system experience. (Note that a rule may be plausible without necessarily being useful of even correct.) The rule discovery system discussed here employs genetic algorithms.

6.2 Some Terms Before Starting BBC

- **Credit**

  Generally the rules in the performance system are of varying usefulness and some, or even most, of them may be incorrect.

  Somehow the system must evaluate the rules. This activity is often called credit assignment (or apportionment of credit); accordingly this level of the system will be called the credit assignment system.

- **Strength of a rule**

  A classifier system uses groups of rules as the representation. The structure of the concept is modeled by the organization, variability, and distribution of strength among the rules. Because the members of a group compete to become active, the appropriate aspects of the representation are selected only when they are relevant in a given problem solving context. The modularity of the concept thereby makes it easier to use as well as easier to modify.

  Basically, strength of a rule tells if the rule is applicable to a particular/ given environment.

- **Classifier** The starting point for this approach to machine learning is a set of rule-based systems suited to rule discovery algorithms. The rules must lend themselves to processes that extract and recombine "building blocks" from currently useful rules to form new rules, and the rules must interact simply and in a highly parallel fashion.

  Classifier systems are parallel, message-passing, rule-based systems wherein all rules have the same simple form. In the simplest version all messages are required to be of a fixed length over a specified alphabet, typically k-bit binary strings. The rules are in the usual condition/ action form. The condition part specifies what kinds of messages satisfy (activate) the rule and the action part specifies what message is to be sent when the rule is satisfied. A classifier system consists of four basic parts.

  1. The input interface translates the current state of the environment into standard messages. For example, the input interface may use property detectors to set the bit values (1: the current state has the property, 0: it does not) at given positions in an incoming message.
  2. The classifiers, the rules used by the system, define the system's procedures for processing messages.
  3. The message list contains all current messages (those generated by the input interface and those generated by satisfied rules).
  4. The output interface translates some messages into effects or actions, actions that modify the state of the environment.

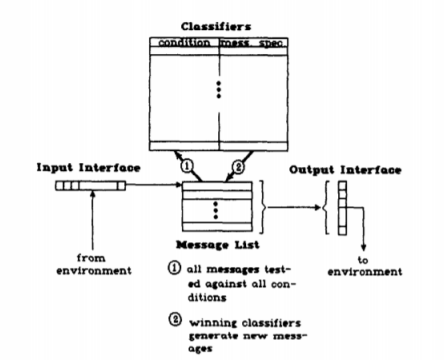


Figure 45: Basic parts of Classifier Systems

A classifier system's basic execution cycle consists of the following steps:

- **Step 1 :** Add all messages from the input interface to the message list.
- **Step 2 :** Compare all messages on the message list to all conditions of all classifiers and record all matches (satisfied conditions).
- **Step 3 :** For each set of matches satisfying the condition part of some classifier, post the message specified by its action part to a list of new messages.
- **Step 4 :** Replace all messages on the message list by the list of new messages.
- **Step 5 :** Translate messages on the message list to requirements on the output interface, thereby producing the system's current output.
- **Step 6 :** Return to Step 1.

Individual classifiers must have a simple, compact definition if they are to serve as appropriate grist for the learning mill; a complex, interpreted definition makes it difficult for the learning algorithm to find and exploit building blocks from which to construct new rules.

The major technical hurdle in implementing this definition is that of providing a simple specification of the condition part of the rule. Each condition must specify exactly the set of messages that satisfies it. Though most large sets can be defined only by an explicit listing, there is one class of subsets in the message space that can be specified quite compactly, the hyperplanes in that space. Specifically, let 1, 0 be the set of possible k-bit messages; if we use "" as a "don't care" symbol, then the set of hyperplanes can be designated by the set of all ternary strings of length k over the alphabet 1, 0, . For example, the string 1... designates the set of all messages that start with a 1, while the string 00... 0 specifies the set 00... 01, 00... 00 consisting of exactly two messages, and so on.

It is easy to check whether a given message satisfies a condition. The condition and the message are matched position by position, and if the entries at all non- positions are identical, then the message satisfies the condition. The notation is extended by

allowing any string c over 1, 0, to be prefixed by a "-" with the intended interpretation that -c is satisfied just in case no message satisfying c is present on the message list.

6.3 Credit Appointment using BBC

The first major learning task facing any rule-based system operating in a complex environment is the credit assignment task. Somehow the performance system must determine both the rules responsible for its successes and the representativeness of the conditions encountered in attaining the successes.The task is difficult because overt rewards are rare in complex environments; the system's behavior is mostly "stage-setting" that makes possible later successes. The problem is even more difficult for parallel systems, where only some of the rules active at a given time may be instrumental in attaining later success. An environment exhibiting perpetual novelty adds still another order of complexity. Under such conditions the performance system can never have an absolute assurance that any of its rules is "correct." The perpetual novelty of the environment, combined with an always limited sampling of that environment, leaves a residue to uncertainty.

Each rule in effect serves as a hypothesis that has been more or less confirmed. The bucket brigade algorithm is designed to solve the credit assignment problem for classifier systems. To implement the algorithm, each classifier is assigned a quantity called its strength. The bucket brigade algorithm adjusts the strength to reflect the classifier's overall usefulness to the system. The strength is then used as the basis of a competition. Each time step, each satisfied classifier makes a bid based on its strength, and only the highest bidding classifiers get their messages on the message list for the next time step. It is worth recalling that there are no consistency requirements on posted messages; the message list can hold any set of messages, and any such set can direct further competition. The only point at which consistency enters is at the output interface. Here, different sets of messages may specify conflicting responses. Such conflicts are again resolved by competition. For example, the strengths of the classifiers advocating each response can be summed so that one of the conflicting actions is chosen with a probability proportional to the sum of its advocates.

The bidding process is specified as follows. Let s(C, t) be the strength of classifier C at time t. Two factors clearly bear on the bidding process:

- relevance to the current situation.
- past "usefulness".

Relevance is mostly a matter of the specificity of the rule's condition part–a more specific condition satisfied by the current situation conveys more information about that situation. The rule's strength is supposed to reflect its usefulness. In the simplest versions of the competition the bid is a product of these two factors, being 0 if the rule is irrelevant (condition not satisfied) or useless (strength 0), and being high when the rule is highly specific to the situation (detailed conditions satisfied) and well confirmed as useful (high strength). To implement this bidding procedure, we modify Step 3 of the basic execution cycle,

For each set of matches satisfying the condition part of classifier C, calculate a bid according to the following formula,

$$B(C,t) = b * R(C)s(C,t) \tag{54}$$

where,

- R(C) is the specificity, equal to the number of non- in the condition part of C divided by the length thereof.
- b is a constant less then one.

The size of the bid determines the probability that the classifier posts its message (specified by the action part) to the new message list. (E.g., the probability that the classifier posts its message might decrease exponentially as the size of the bid decreases.)

The use of probability in the revised step assures that rules of lower strength sometimes get tested, thereby providing for the occasional testing of less favored and newly generated (lower strength) classifiers ("hypotheses").

The operation of the bucket brigade algorithm can be explained informally via an economic analogy. The algorithm treats each rule as a kind of "middleman" in a complex economy. As a "middleman," a rule only deals with its "suppliers"–the rules sending messages satisfying its conditions–and its "consumers"–the rules with conditions satisfied by the messages the "middleman" sends. Whenever a rule wins a bidding competition, it initiates a transaction wherein it pays out part of its strength to its suppliers. (If the rule does not bid enough to win the competition, it pays nothing.) As one of the winners of the competition, the rule becomes active, serving as a supplier to its consumers, and receiving payments from them in turn. Under this arrangement, the rule's strength is a kind of capital that measures its ability to turn a "profit." If a rule receives more from its consumers than it paid out, it has made a profit; that is, its strength has increased.

More formally, when a winning classifier C places its message on the message list it pays for the privilege by having its strength s(C, t) reduced by the amount of the bid B(C, t),

$$s(C,t+1) = s(C,t) - B(C,t). \tag{55}$$

The Classifiers C' sending messages matched by this winner, the "suppliers," have their strengths increased by the amount of the bid–it is shared among them in the simplest version

$$s(C',t+1) = s(C',t) + aB(C,t), \tag{56}$$

where,

- $a = \dfrac{1}{(|C'|)}$

A rule is likely to be profitable only if its consumers, in their local transactions, are also (on the average) profitable. The consumers, in turn, will be profitable only if their consumers are profitable. The resulting chains of consumers lead to the ultimate consumers, the rules that directly attain goals and receive payoff directly from the environment. (Payoff is added to the strengths of all rules determining responses at the time the payoff occurs.) A rule that regularly attains payoff when activated is of course profitable. The profitability of other rules depends upon their being coupled into sequences leading to these profitable ultimate consumers. The bucket brigade ensures that early acting, "stage-setting" rules eventually receive credit if they are coupled into (correlated with) sequences that (on average) lead to payoff. If a rule sequence is faulty, the final rule in the sequence loses strength, and the sequence will begin to disintegrate, over time, from the final rule backwards through its chain of precursors. As soon as a rule's strength decreases to the point that it loses in the bidding process, some competing rule will get a chance to act as a replacement. If the competing rule is more useful than the one displaced, a revised rule sequence will begin to form using the new rule. The bucket brigade algorithm thus searches out and repairs "weak links" through its pervasive local application.

Whenever rules are coupled into larger hierarchical knowledge structures, the bucket brigade algorithm is still more powerful than the description so far would suggest. Consider an abstract rule C* of the general form, "if the goal is G, and if the procedure P is executed, then G will be achieved." C* will be active throughout the time interval in which the sequence of rules comprising P is executed. If the goal is indeed achieved, this rule serves to activate the response that attains the goal, as well as the stage-setting responses preceding that response. Under the bucket brigade C* will be strengthened immediately by the goal attainment. On the very next trial involving P, the earliest rules in P will have their strengths substantially increased under the bucket brigade. This happens because the early rules act as suppliers to the strengthened C* (via the condition "if the procedure P is executed"). Normally, the process would have to be executed on the order of n times to backchain strength through an n-step process P. C* circumvents this necessity.

6.4 Usage of BBC in Decision Tree

Decision Trees have one main flaw that prevents them from being the ideal tool for predictive learning, and that is inaccuracy. In other words, they are great to work with the data used to create them, but they aren't flexible when it comes to classifying new samples.

One way to overcome this is to create many decision trees by taking a random subsets of the of attributes and sample points.

Now, we can when we have a new data point, we proceed by passing it into all of the decision trees and classify it aggregate of all the decisions made by the decision tree.

But there is one flaw to this method, and that is to give equal weightage to all the decision tree irrespective of the input.

This is where we can use BBC, where the credit would be how much weightage the tree has on the decision and strength of the tree to be a probabilistic measure of how likely the tree is suitable for the input.

This way, we bring in BBC concepts into decision tree to make it more flexible.