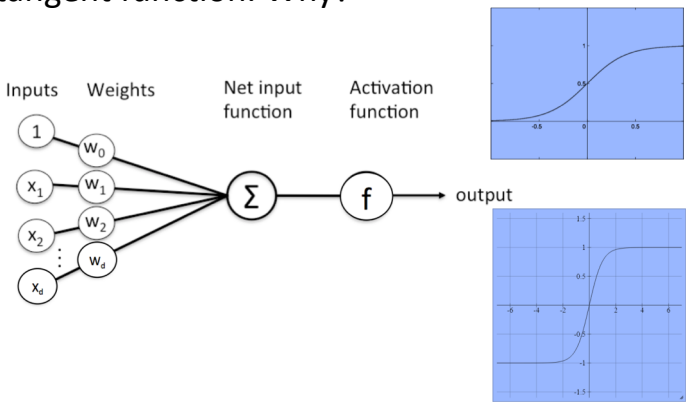
Building Classifiers: Part 4

Ishwar K Sethi

Sigmoidal Neuron

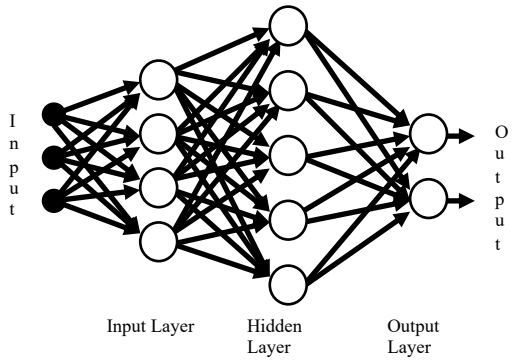
- The activation function of neurons in a multi-layer neural network needs to be a smooth function, e.g. sigmoid function or hyperbolic tangent function. Why?



Output = 1/(1+ exp(-net))

Output = (1- exp(-net))/(1+ exp(-net)

Multiple-Layer Feedforward Network Model

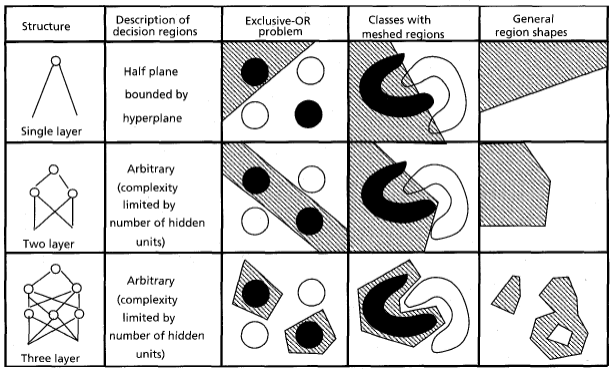


One can use many hidden layers.

Multiple-Layer Feedforward Network Model

- Capable of generating arbitrarily complex models
- Error-minimization learning (Backpropagation learning)
- Long training time
- Performance affected by network size

A geometric interpretation of the role of hidden unit in a two-dimensional input space.



There is a mathematical proof that states that any arbitrary function can be approximated with a single hidden layer network. Hence neural nets are called *universal approxima*tors.

How Do We Train a Multilayer Neural Network?

- In supervised learning, we are given a collection of vector pairs $\{ \mathbf{x}_i, \mathbf{y}_i \}$
- Each $\mathbf{x}_i$ is a vector of real numbers of some suitable dimensionality
- Each $\mathbf{y}_i$ is a vector with components as 0 or 1 (when sigmoidal activation function is used) or -1 or 1 (when tanh is used). When the network has a single output, then each $\mathbf{y}_i$ is a scalar
- We can set up a loss function at the output and use gradient descent to adjust the weights at the output layer to minimize the loss function
- The gradient at the output is fed back to internal layers to adjust their weights. This feeding back of the gradient to internal layers is based on chain-rule of differentiation and the resulting algorithm is known as backpropagation.

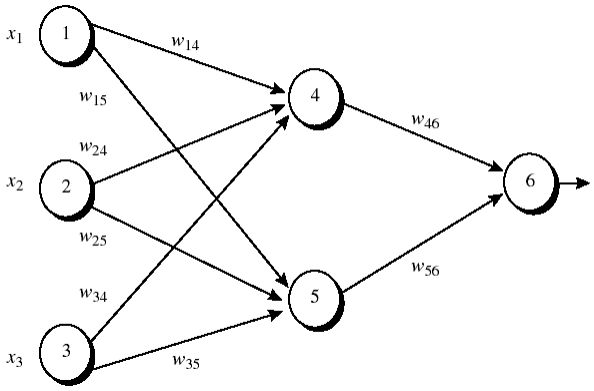
MLP Training

- Works in two phases
  - Feedforward phase
  - Backpropagation phase
- During test/usage phase, only feedforward phase is needed to produce the output

Feedforward Operation

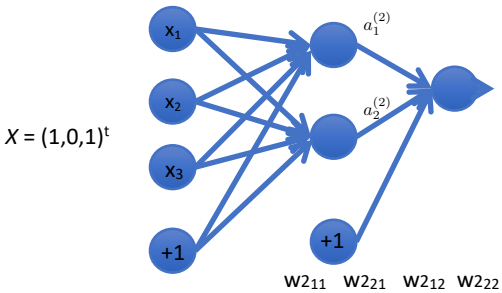
- Each hidden unit computes the weighted sum of its inputs to form a net activation value, $net = \mathbf{w}^T\mathbf{x}$.
- Then, it emits an output that is a nonlinear function of its activation, $f(net)$.
- Each output unit similarly computes its net activation based on the hidden unit signals in the previous layer, and emits a value using a nonlinear function based on its activation.
- The output corresponds to a nonlinear function of the input feature vector.

An Example



- Assume that the learning rate η is 0.9 and the first training example is $X = (1,0,1)^t$ with the desired output of 1.

An Example of Forward Propagation



Weight values

w_{211}	w_{221}	w_{212}	w_{222}	w_{231}	w_{232}	w_{311}	w_{356}	b_{11}	b_{12}	b_{13}
0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

[Forward Propagation]

Unit	Net input	Output
(12)	0.2*1 + 0.4*0 -0.5 *1-0.4*1 = -0.7	$1/(1+e^{0.7})=0.332$
(22)	-0.3*1 +0.1*0+0.2 *1+0.2*1 =0.1	$1/(1+e^{-0.1})=0.525$
(13)	-0.3*0.332 -0.2*0.525 +0.1*1 = -0.105	
	$1/(1+e^{0.105})=0.474$	

Backpropagation

- *Backpropagation* is one of the simplest and most general methods for supervised training of multilayer neural networks.
- It is based on the least-mean-square algorithm where the error is proportional to the square of the difference between the actual output and the desired output.
- The dependence of the error on the hidden-to-output layer weights is straightforward.
- The backpropagation algorithm allows us to calculate an effective error for each hidden unit and derive a learning rule for the input-to-hidden weights.

Chain Rule of Differentiation

- The heart of backpropagation algorithm is the chain rule of differentiation.

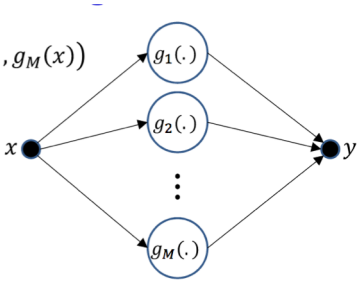
$$f(y) = sin(y),\ y = g(x) = 0.5x^2$$

$$\frac{df}{dx} = ?$$

$$\frac{df}{dx} = \frac{df}{dy}\frac{dy}{dx} = cos(0.5x^2)\ x$$

Chain Rule of Differentiation

$y = f(g_1(x), g_1(x), \ldots, g_M(x))$



- x affects y through each of $g_1 \ldots g_M$

$$\frac{dy}{dx} = \frac{\partial y}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial y}{\partial g_M(x)}\frac{dg_M(x)}{dx}$$

Backpropagation Algorithm

- For the hidden-to-output weights, the weight update rule becomes

δ_k:Sensitivity of the k-th output unit

$$\Delta w_{kj} = -\eta \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = \eta(t_k - z_k)f'(net_k)y_j$$

where net_k is the net activation value.

- For the input-to-hidden weights, the weight update rule becomes

For a unit with the sigmoidal activation function: *f'(net)=z(1-z)*

$$\Delta w_{ji} = -\eta \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

$$= \eta \left(\sum_{k=1}^{c} w_{kj}(t_k - z_k)f'(net_k) \right) f'(net_j)x_i.$$

δ_j:Sensitivity of the j-th hidden unit

Backpropagation Algorithm

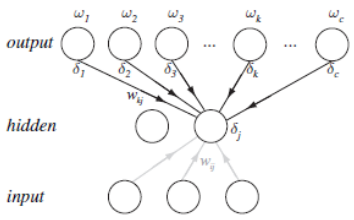


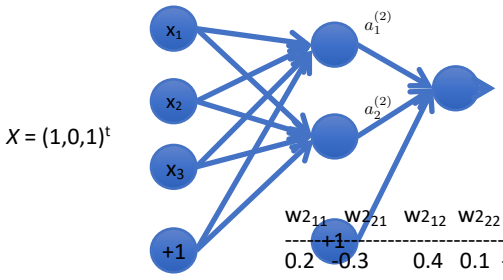
FIGURE 6.5. The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Credit assignment problem solved by generalized delta rule

Backpropagation Algorithm

- The algorithm is called "backpropagation" because it propagates the error from the output layer back to the hidden layers.
- As with all gradient descent procedures, the exact behavior of the backpropagation algorithm depends on the starting point.
- The initial weights are randomly chosen from a uniform distribution symmetric around zero.

Backward Pass Computation Example



$X = (1,0,1)^t$

Weight values

$w2_{11}$	$w2_{21}$	$w2_{12}$	$w2_{22}$	$w2_{13}$	$w2_{23}$	$w3_{11}$	$w3_{12}$	$b1_1$	$b1_2$	$b1_3$
0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

[Forward Propagation]

Unit	Net input	Output
(12)	0.2*1 + 0.4*0 -0.5 *1-0.4*1 = -0.7	$1/(1+e^{0.7})=0.332$
(22)	-0.3*1 +0.1*0+0.2 *1+0.2*1 =0.1	$1/(1+e^{-0.1})=0.525$
(13)	-0.3*0.332 -0.2*0.525 +0.1*1 = -0.105	
	$1/(1+e^{0.105})=0.474$	

We saw this earlier

Calculated deltas

(13) (0.474)(1-0.474)(1-0.474)=0.1311
(22) (0.525)(1-0.525)(0.1311)(-0.2)=-0.0065
(12) (0.332)(1-0.332)(0.1311)(-0.3)=-0.0087

Calculations for weight updating

Weight	New value
w_{46}	-0.3+(0.9)(0.1311)(0.332)= -0.261
w_{56}	-0.2+(0.9)(0.1311)(0.525)= -0.138
w_{14}	0.2 +(0.9)(-0.0087)(1) = 0.192
w_{15}	-0.3 +(0.9)(-0.0065)(1) = -0.306
w_{24}	0.4+ (0.9)(-0.0087)(0) = 0.4
w_{25}	0.1+ (0.9)(-0.0065)(0) = 0.1
w_{34}	-0.5+ (0.9)(-0.0087)(1) = -0.508
w_{35}	0.2 + (0.9)(-0.0065)(1) = 0.194
w_{06}	0.1 + (0.9)(0.1311) = 0.218
w_{05}	0.2 + (0.9)(-0.0065)=0.194
w_{04}	-0.4 +(0.9)(-0.0087) = -0.408

Practical Issues in BP Training: When to Stop Training?

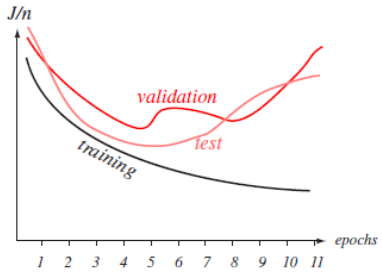


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Practical Issues in Training: Initializing Weights

- The weights are initialized randomly but it is a good practice to choose hidden unit weights in the range of [-1/sqrt(*d*) 1/sqrt(*d*)], *d*: dimensionality of the input vectors
- A good choice for the weights of the output units is the range [-1/sqrt(# of hidden units) 1/sqrt(# of hidden units)]
- The above choices for initialization ensure
  - All units learn at same pace
  - Learning typically will begin from a state where each unit is operating in the linear part of its activation function

Practical Issues in Training: Number of Hidden Units

- The number of hidden units influences learning rate and generalization accuracy
- Fewer hidden units imply relatively simpler decision boundaries. So in some sense the complexity of the problem determines the number of hidden units needed
- More hidden units mean more weights to be determined and hence more training data
- A popular rule of thumb is to choose the number of hidden units such that there are roughly 10 training examples per weight

Practical Issues in Training: Use of Momentum Term

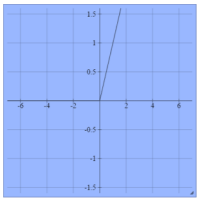
- Often a momentum term is added to weight change rule to speed up learning

$$\boldsymbol{w}(m+1) = \boldsymbol{w}(m) + (1-\alpha)\Delta\boldsymbol{w}_{bp}(m) + \alpha\Delta\boldsymbol{w}(m-1)$$

Weight change due to the backpropagation algorithm

The momentum term reflecting the previous weight update. Alpha is 0 for no momentum.

Currently Favored Activation Function



Rectified Linear Unit (ReLU) Function

Output = net if *net* > 0
= 0 if *net* < 0
Or Output = max(0, net)

This function has become popular for neurons. Empirical results have shown that training convergence is 5-6 times faster using this function compared to tanh function. Computationally it is simpler. However, ReLU usage can cause part of the network to result in dead units because linear part of the function can drive some units to operate in the left half part of ReLU function. Thus, it requires greater care in choosing the learning rate.

Cross Entropy Criterion

Used for multi-class problems. Let C stand for a random variable representing class label of a training example. Let $p(c_i)$ stand for the probability of label being c_i. Then the entropy of the training set is

$$H(C) = -\sum_i p(c_i) \log p(c_i)$$

Lets say our training data has four labels: cat, dog, horse, sheep. Let the mix of labels in our training data be: cat 40%, dog 10%, horse 25%, sheep 25%. Then

Entropy of our training set *H = -(0.4 log0.4* + 0.1log0.1 + 0.25log0.25 + 0.25log0.25 = 1.86

Entropy is a measure of uncertainty. What will happen if all labels have equal representation in the mix?

Cross Entropy Criterion

Now consider two neural networks, Net1 and Net2.
Net1 produces the following classification result on our training data:
Cat 25%, dog 25%, horse 25%, and sheep 25%
Net2 yields the following result:
Cat 40%, dog 10%, horse 10%, sheep 40%

How do we say whether Net1 is better or Net 2 is better?
By cross entropy

$$H(C, \hat{C}) = -\sum_i p(c_i) \log p(\hat{c}_i)$$

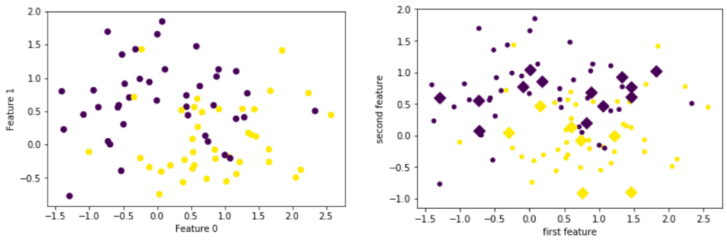
$\hat{C}$ stands for label distribution in the classifier output

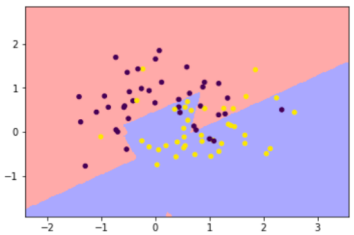
H(C, Net1) = 2.0 *H(C, Net2)* = 2.02

Net1 is slightly better because cross entropy in this case, 2.0, is closer to the entropy of the training set, 1.86

Cross entropy is a measure to determine how similar two probability distributions are. It is not a symmetric function. The loss function surface with this criterion is considered better for gradient search.

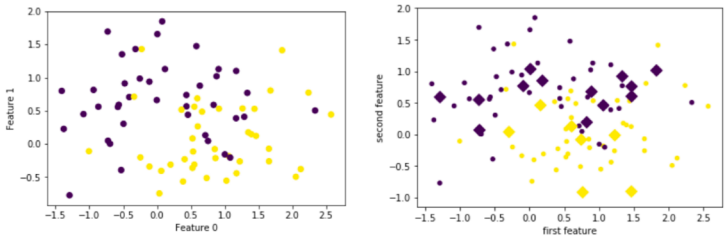
MLP Training Example

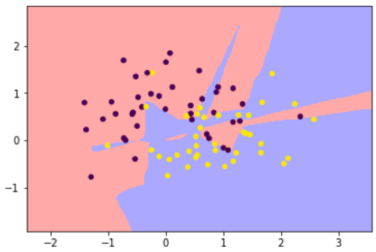




Hidden layer size = 5
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 84%
Test accuracy = 75%

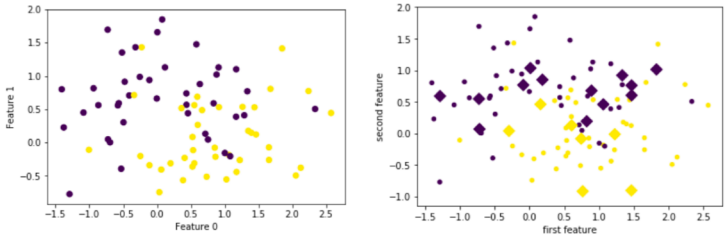
MLP Training Example

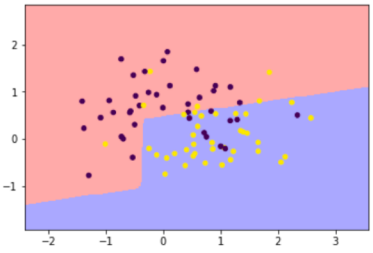




Hidden layer size = 10
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 95%
Test accuracy = 90%

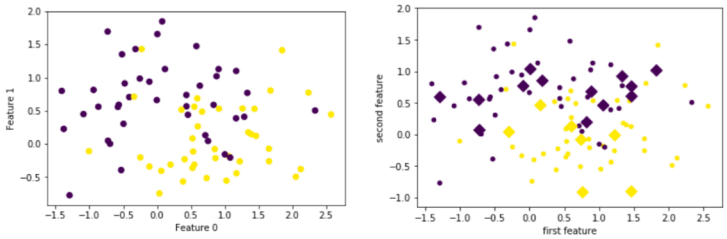
MLP Training Example

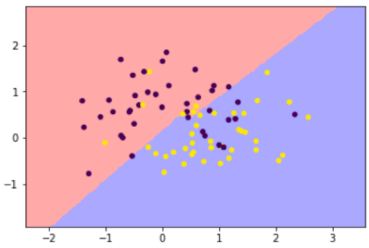




Hidden layer size = 2
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 81%
Test accuracy = 70%

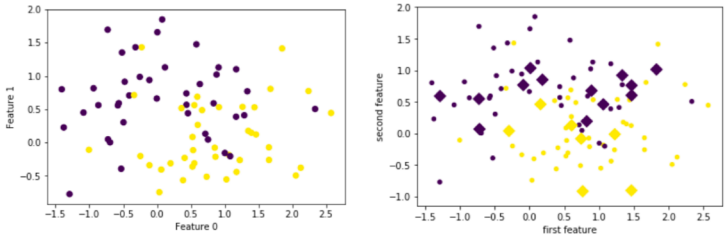
MLP Training Example

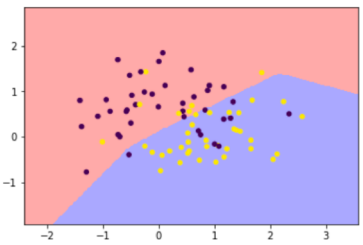




Hidden layer size = 5
Learning rate = 0.05
Activation Function = Sigmoidal
Training Accuracy = 81%
Test accuracy = 60%

MLP Training Example





Hidden layer size = 5
Learning rate = 0.05
Activation Function = Relu
Training Accuracy = 79%
Test accuracy = 70%

Effect of Normalization

```
features, target = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(features, target,
                                                    test_size=0.30,
                                                    random_state=15)
```

```
mlp = MLPClassifier(hidden_layer_sizes =75,activation = 'tanh',learning_rate_init=0.01,batch_size=10,solver='lbfgs', random_state=0).fit(X_train, y_train)
result = mlp.predict(X_test)
```

```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train, y_train)))
y_pred = mlp.predict(X_test)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test, y_test)))
y_pred = mlp.predict(X_test)
```

```
Training_accuracy: 0.92
Test_accuracy: 0.81
```

```
scalar = StandardScaler()
scalar.fit(X_train)
X_train_scaled=scalar.transform(X_train)
X_test_scaled = scalar.transform(X_test)
```

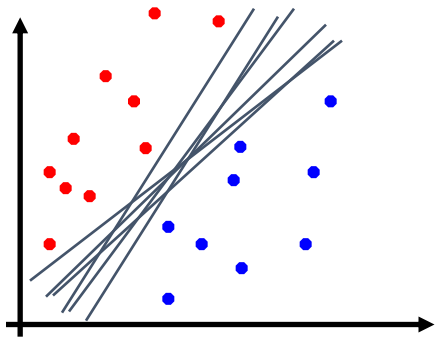
```
mlp = MLPClassifier(hidden_layer_sizes =75,activation = 'tanh',learning_rate_init=0.01,batch_size=10,solver='lbfgs', random_state=0).fit(X_train_scaled, y_train)
result = mlp.predict(X_test_scaled)
```

```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train_scaled, y_train)))
y_pred = mlp.predict(X_test_scaled)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test_scaled, y_test)))
y_pred = mlp.predict(X_test_scaled)
```

```
Training_accuracy: 1.00
Test_accuracy: 0.98
```

Support Vector Machine (SVM)

- Which of the linear boundaries is better?



Support Vector Machine (SVM)

- Why not look for a "fat line" to provide for some margin across the decision boundary?

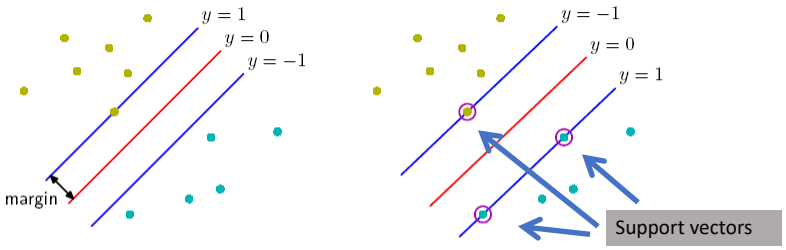
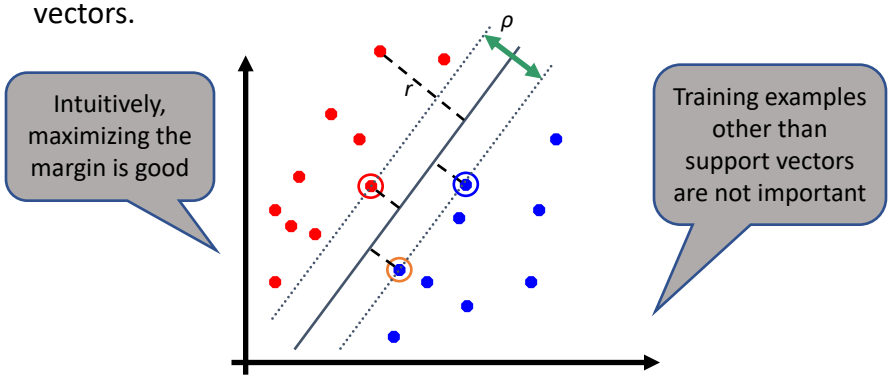


Figure 5: The margin is defined as the perpendicular distance between the decision boundary and the closest of the data points (left). Maximizing the margin leads to a particular choice of decision boundary (right). The location of this boundary is determined by a subset of the data points, known as the support vectors, which are indicated by the circles.

Support Vector Machine (SVM)

- Distance from example $\mathbf{x}_i$ to the separator is $r = \frac{\mathbf{w}^T\mathbf{x}_i + w_0}{\|\mathbf{w}\|}$
- Examples closest to the hyperplane are ***support vectors***.
- ***Margin*** ρ of the separator is the distance between support vectors.



Nonlinear SVM

- Most interesting problems are nonlinear
- Use a mapping to a higher dimensional space wherein the boundary may be linear
- The linear classifier relies on inner product between vectors $K(\mathbf{x}_i,\mathbf{x}_j)=\mathbf{x}_i^T\mathbf{x}_j$
- If every datapoint is mapped into high-dimensional space via some transformation Φ: $\mathbf{x} \rightarrow \phi(\mathbf{x})$, the inner product becomes:

$$K(\mathbf{x}_i,\mathbf{x}_j)= \boldsymbol{\phi}(\mathbf{x}_i)^T\boldsymbol{\phi}(\mathbf{x}_j)$$

An Example

- Consider $X^t = [2\ 3]$, $Y^t = [1\ 2]$
- X^tY is then 8 and $(X^tY)^2$ is 64
- Now consider a 3-d space where X is given as $[4\ \sqrt{2}*6\ 9]^t$ and Y as $[1\ \sqrt{2}*2\ 4]^t$
- The relationship between original X and Y and the mapped X and Y is $(a_1^2\ \sqrt{2}a_1a_2\ a_2^2)$ for any point $(a_1\ a_2)$ in the original space
- The dot product of X and Y in the new space is 64 which is same as $(X^tY)^2$
- Let us try this for another X-Y pair, say $X^t = [5\ 2]$, $Y^t = [2\ 7]$
- X^tY is then 24 and $(X^tY)^2$ is 576
- 3-d representations of X and Y are $[25\ \sqrt{2}*10\ 4]^t$ and $[4\ \sqrt{2}*14\ 49]^t$, respectively
- Dot product in this space is again square of the dot product in the original space
- Thus, we can compute the dot product in the mapped space without explicit mapping of the original data into higher space. This property is not true for any arbitrary mappings but only for certain mappings defined by functions satisfying certain properties. Such functions are known as kernel functions and this shortcut to computation is called kernel trick.

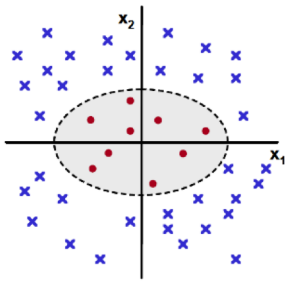
What is a Kernel Function?

- A function that takes as its inputs vectors in the original space and returns the dot product of the vectors in the feature space is called a *kernel function*.
- In other words, a kernel function *k(x,y)*
  - is a similarity measure
  - defined by an implicit mapping ϕ,
  - from the original space to a vector space (feature space)
  - such that: $k(x,y)=\phi(x)\bullet\phi(y)$
- A function *k(x,y)* is a valid kernel if *k* is positive definite and symmetric (Mercer Theorem)
- Symmetric → *k(x,y) = k(y,x)*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j k(\mathbf{x}_i \mathbf{x}_j) \geq 0$$

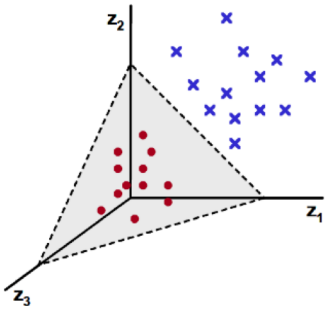
c_i and c_j positive numbers

Kernel Example (1)



Original two-dimensional space with patterns from two classes

Mapped three-dimensional space with $z_1=x_1^2$, $z_2=\sqrt{2}x_1x_2$, and $z_3=x_2^2$. The data is linearly separable in this space.



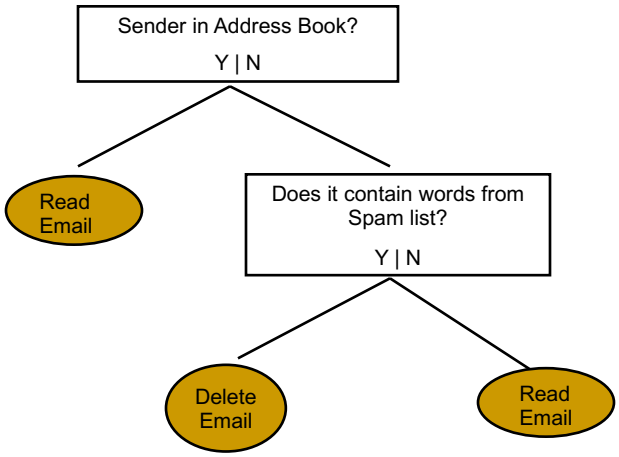
SVM in Practice

- SVMs are currently among the best performers for a number of classification tasks ranging from text to genomic data.
- Most popular optimization algorithms for SVMs use *decomposition* to hill-climb over a subset of α_i's at a time, e.g. SMO [Platt '99] and [Joachims '99]
- Tuning SVMs remains a black art: selecting a specific kernel and parameters is usually done by trials

Decision Tree Classifiers (DCT)

- The classifier has a tree structure where each node is either:
  - A leaf node with a class label, or
  - An internal node indicating some test to be carried out on the example passing through it
- Most tree classifiers use a single attribute based test at internal nodes

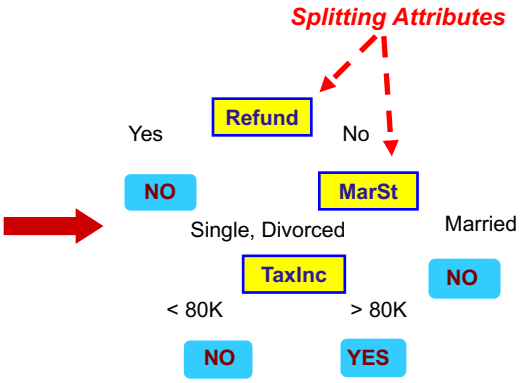
A Simple DTC Example



Another Example of a Decision Tree

Tid	Refund	Marital Status	Taxable Income	Cheat
	categorical	categorical	continuous	class
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Training Data



Building Decision Trees

- A two phase procedure consisting of growing and pruning
- The growth phase searches for successive splits to divide training examples into smaller subsets of increasing purity or homogeneity in a top-down greedy manner
- Splitting is generally done by finding an attribute-value pair that maximizes some purity criterion

Decision Tree Induction

- Many Algorithms:
  - AMIG
  - CART
  - ID3, C4.5
  - Perceptron Tree

Tree Induction

- Issues in tree growing
  - Determine how to split the records
    - How to specify the attribute test condition?
    - How to determine the best split?
  - Determine when to stop splitting

How to Specify A Test Condition?

- Depends on attribute types
  - Nominal
  - Ordinal
  - Continuous
- Depends on number of ways to split
  - 2-way split
  - Multi-way split

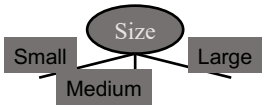
Splitting Based on Nominal Attributes

- Multi-way split: Use as many partitions as distinct values.

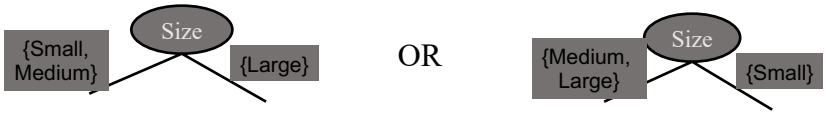
- Binary split: Divides values into two subsets. Need to find optimal partitioning.

Splitting Based on Ordinal Attributes

- Multi-way split: Use as many partitions as distinct values.



- Binary split: Divides values into two subsets. Need to find optimal partitioning.



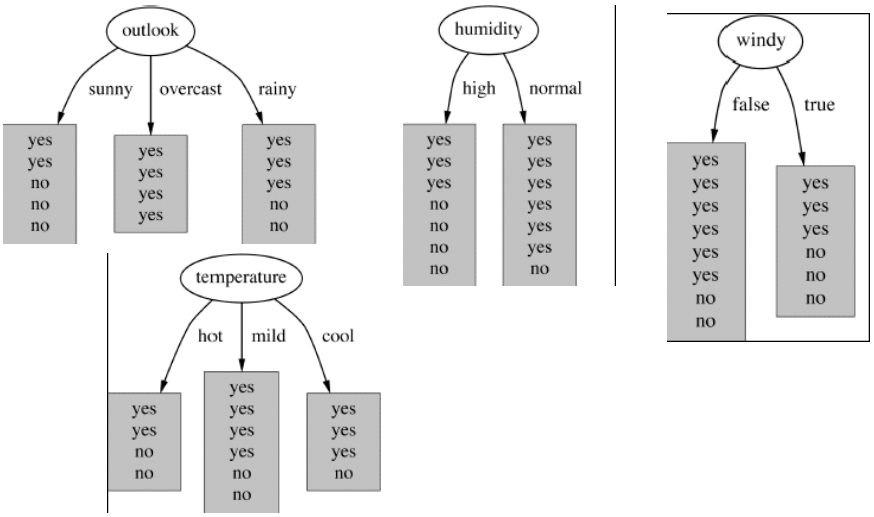
Splitting Based on Continuous Attributes

- Different ways of handling
  - Discretization to form an ordinal categorical attribute
    - Static – discretize once at the beginning
    - Dynamic – ranges can be found by equal interval bucketing, equal frequency bucketing (percentiles), or clustering.
  - Binary Decision: $(A < v)$ or $(A \geq v)$
    - consider all possible splits and finds the best cut

A Simple Example of Tree Building

Outlook	Temp	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Which attribute to select?



Measuring Split Quality

- Select an attribute that is likely to result in a small tree
- Small tree is likely to result from splits that are "purer"
- Use some kind of purity measure

Information Gain Based Measure

$$\text{entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \ldots - p_n \log p_n$$

Entropy of attribute Outlook

Outlook = *Sunny*:

$\text{info}([2,3]) = \text{entropy}(2/5, 3/5) = -2/5 \log(2/5) - 3/5 \log(3/5) = 0.971 \text{ bits}$

Outlook = *Overcast*:

$\text{info}([4,0]) = \text{entropy}(1,0) = -1 \log(1) - 0 \log(0) = 0 \text{ bits}$

Note: this is normally undefined.

Outlook = *Rainy*:

$\text{info}([2,3]) = \text{entropy}(3/5, 2/5) = -3/5 \log(3/5) - 2/5 \log(2/5) = 0.971 \text{ bits}$

Expected information for attribute:

$\text{info}([3,2],[4,0],[3,2]) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971 = 0.693 \text{ bits}$

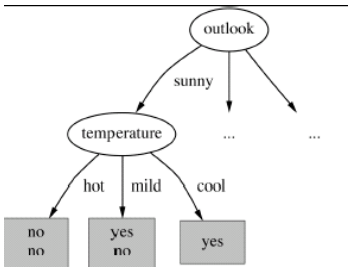
Information Gains Based on Different Splits

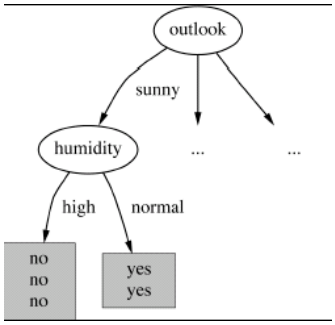
- Information gain: information before splitting – information after splitting

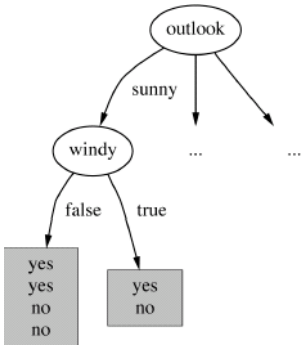
  gain(*Outlook*) = info([9,5]) – info([2,3],[4,0],[3,2])
  = 0.940 – 0.693
  = 0.247 bits

- Information gain for attributes from weather data:

  gain(*Outlook*) = 0.247 bits
  gain(*Temperature*) = 0.029 bits
  gain(*Humidity*) = 0.152 bits
  gain(*Windy*) = 0.048 bits

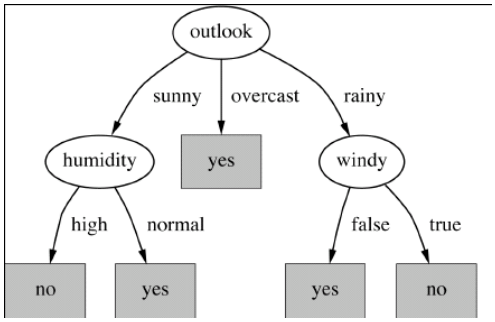






gain(*Temperature*) = 0.571 bits
gain(*Humidity*) = 0.971 bits
gain(*Windy*) = 0.020 bits

Final Decision Tree



Note: not all leaves need to be pure; sometimes identical instances have different classes
⇒ Splitting stops when data can't be split any further

Finding Splits on Numerical Attributes

- Split on temperature attribute:

64	65	68	69	70	71	72	72	75	75	80	81	83	85
Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	No

  - E.g. temperature < 71.5: yes/4, no/2
    temperature ≥ 71.5: yes/5, no/3
  - Info([4,2],[5,3])
    = 6/14 info([4,2]) + 8/14 info([5,3])
    = 0.939 bits
- Place split points halfway between values
- Can evaluate all split points in one pass!

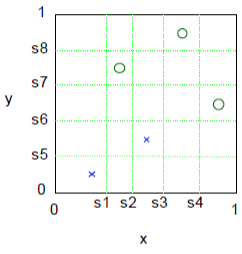
Entropy Measure for Splits' Quality: Continuous Attributes

$$E = -\sum_{j}^{M} p_j \ln p_j$$

$$E = -\frac{2}{5}\ln(\frac{2}{5}) - \frac{3}{5}\ln(\frac{3}{5}) = 0.6730$$

$$\Delta E(s) = E - p_{left}E_{left} - p_{right}E_{right}$$

o: class 1, x: class 2



To see which split should be preferred, let us calculate the change in impurity due to each split. We show these calculations for split s_2.

$$p_{left} = \frac{2}{5},$$

$$E_{left} = -\frac{1}{2}\ln(\frac{1}{2}) - \frac{1}{2}\ln(\frac{1}{2}) = 0.6983,$$

$$p_{right} = \frac{3}{5},$$

$$E_{right} = -\frac{1}{3}\ln(\frac{1}{3}) - \frac{2}{3}\ln(\frac{2}{3}) = 0.6365$$

$$\Delta E(s_1) = 0.2231,$$
$$\Delta E(s_3) = 0.2911,$$
$$\Delta E(s_4) = 0.1185,$$
$$\Delta E(s_5) = 0.2231,$$
$$\Delta E(s_6) = 0.6730,$$
$$\Delta E(s_7) = 0.2911,$$
$$\Delta E(s_8) = 0.1185.$$

Best split: S_6

Therefore, the change in impurity due to split s_2 is given as

$$\Delta E(s_2) = 0.6730 - 0.4 * 0.6983 - 0.6 * 0.6365$$
$$= 0.0138$$

GINI Measure of Impurity

- Gini Index for a given node t :

$$GINI(t) = 1 - \sum_j [p(j \mid t)]^2$$

(NOTE: $p(j \mid t)$ is the relative frequency of class j at node t).

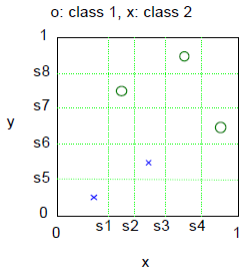
- Maximum (1 - $1/n_c$) when records are equally distributed among all classes, implying least interesting information
- Minimum (0.0) when all records belong to one class, implying most interesting information

GINI Measure of Impurity

$$G = 1 - \sum_{j=1}^{M} p_j^2.$$

$$\Delta G(s) = G - p_{left} G_{left} - p_{right} G_{right}$$

$$G = 1 - (\frac{2}{5})^2 - (\frac{3}{5})^2 = 0.480.$$



$$\Delta G(s_1) = 0.180,$$
$$\Delta G(s_2) = 0.427,$$
$$\Delta G(s_3) = 0.214,$$
$$\Delta G(s_4) = 0.080,$$
$$\Delta G(s_5) = 0.180,$$
$$\Delta G(s_6) = 0.480,$$
$$\Delta G(s_7) = 0.214,$$
$$\Delta G(s_8) = 0.080.$$

Not surprisingly, the best split is s_6.

Stopping Criteria for Tree Induction

- Stop expanding a node when all the records belong to the same class
- Stop expanding a node when all the records have similar attribute values

Pruning for Right Size Tree

- Prevent overfitting to noise in the data
- "Prune" the decision tree
- Two strategies:
  - *Postpruning*
    take a fully-grown decision tree and discard unreliable parts
  - *Prepruning*
    stop growing a branch when information becomes unreliable
- Postpruning preferred in practice—prepruning can "stop early"

Multifeature Splits

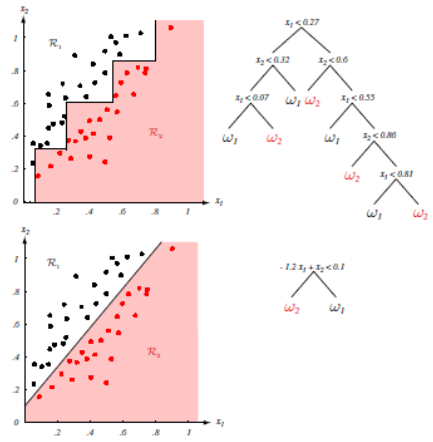


FIGURE 8.5. If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If, however, "proper" decision forms are used (here, linear combinations of the features), the tree can be quite simple, as shown at the bottom. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

DTC Strengths

- Intuitive, easy to comprehend
- Fast classification
- Can deal with continuous and categorical attributes
- Built in attribute selection

DTC Weaknesses

- Prone to errors due to noise in attributes
- May not yield good performance with single feature splits
- Can over-fit if not pruned properly
- May mask important relationships because of greedy top-down search while building the tree

Summary of Classifier Building Methods

- Both linear and nonlinear models can be built
- Overfitting needs to be avoided
- Hyper-parameter tuning may be needed
- Most methods discussed can also be used for regression