Top 9 Data Science Algorithms

An algorithm is a step by step method of solving a problem. It is commonly used for data processing, calculation and other related computer and mathematical operations. In this article, we have listed some of the most important algorithms in the history of Data Science. We've compiled these algorithms based on recommendations by big data enthusiasts on various social media channels.

Contents

Apriori algorithm

Seminar of Popular Algorithms in Data Mining and Machine Learning, TKK

Presentation 12.3.2008

Lauri Lahti

Association rules

- Techniques for data mining and knowledge discovery in databases

Five important algorithms in the development of association rules (Yilmaz et al., 2003):

- AIS algorithm 1993
- SETM algorithm 1995
- Apriori, AprioriTid and AprioriHybrid 1994

Apriori algorithm

- Developed by Agrawal and Srikant 1994
- Innovative way to find association rules on large scale, allowing implication outcomes that consist of more than one item
- Based on minimum support threshold (already used in AIS algorithm)
- Three versions:
  - Apriori (basic version) faster in first iterations
  - AprioriTid faster in later iteratons
  - AprioriHybrid can change from Apriori to AprioriTid after first iterations

Limitations of Apriori algorithm

- Needs several iterations of the data
- Uses a **uniform** minimum support threshold
- Difficulties to find rarely occuring events
- Alternative methods (other than appriori) can address this by using a **non-uniform** minimum support thresold
- Some competing alternative approaches focus on **partition** and **sampling**

Phases of knowledge discovery

1) data selection
2) data cleansing
3) data enrichment (integration with additional resources)
4) data transformation or encoding
5) data mining
6) reporting and display (visualization) of the discovered knowledge

(Elmasri and Navathe, 2000)

Application of data mining

- Data mining can typically be used with transactional databases (for ex. in shopping cart analysis)
- Aim can be to build association rules about the shopping events
- Based on **item sets**, such as

  {milk, cocoa powder} 2-itemset

  {milk, corn flakes, bread} 3-itemset

Association rules

- Items that occur often together can be associated to each other
- These together occuring items form a **frequent itemset**
- Conclusions based on the frequent itemsets form **association rules**
- For ex. {milk, cocoa powder} can bring a rule *cocoa powder* ➔ *milk*

Sets of database

- Transactional database D
- All products an itemset $I = \{i_1, i_2, \ldots, i_m\}$
- Unique shopping event $T \subseteq I$
- T contains itemset X iff $X \subseteq T$
- Based on itemsets X and Y an association rule can be $X \rightarrow Y$
- It is required that $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$

Properties of rules

- Types of item values: boolen, quantitative, categorical
- Dimensions of rules
  - 1D: *buys(cocoa powder)* ➔ *buys(milk)*
  - 3D: *age(X,'under 12')* $\wedge$ *gender(X,'male')* ➔ *buys(X,'comic book')*
- Latter one is an example of a profile association rule
- Intradimension rules, interdimension rules, hybrid-dimension rules (Han and Kamber, 2001)
- Concept hierarchies and multilevel association rules

Quality of rules

- Interestingness problem (Liu et al., 1999):
  - some generated rules can be self-evident
  - some marginal events can dominate
  - interesting events can be rarely occuring
- Need to estimate how interesting the rules are
- Subjective and objective measures

Subjective measures

- Often based on earlier user experiences and beliefs
- Unexpectedness: rules are interesting if they are unknown or contradict the existing knowledge (or expectations).
- Actionability: rules are interesting if users can get advantage by using them
- Weak and strong beliefs

Objective measures

- Based on threshold values controlled by the user
- Some typical measures (Han and Kamber, 2001):
  - simplicity
  - support (utility)
  - confidence (certainty)

Simplicity

- Focus on generating simple association rules
- Length of rule can be limited by user-defined threshold
- With smaller itemsets the interpretation of rules is more intuitive
- Unfortunately this can increase the amount of rules too much
- Quantitative values can be quantized (for ex. age groups)

Simplicity, example

- One association rule that holds association between cocoa powder and milk

  buys(cocoa powder) ➔ *buys(bread,milk,salt)*
- More simple and intuitive might be

  buys(cocoa powder) ➔ *buys(milk)*

Support (utility)

- Usefulness of a rule can be measured with a minimum support threshold
- This parameter lets to measure how many events have such itemsets that match both sides of the implication in the association rule
- Rules for events whose itemsets do not match boths sides sufficiently often (defined by a threshold value) can be excluded

Support (utility) (2)

- Database D consists of events T_1, T_2,… T_m, that is $D = \{T_1, T_2, \ldots, T_m\}$
- Let there be an itemset X that is a subregion of event T_k, that is $X \subseteq T_k$
- The support can be defined as

$$sup(X) = \frac{|\{T_k \in D \mid X \subseteq T_k\}|}{|D|}$$

- This relation compares number of events containing itemset X to number of all events in database

Support (utility), example

- Let's assume D = {(1,2,3), (2,3,4), (1,2,4), (1,2,5), (1,3,5)}
- The support for itemset (1,2) is

$$\sup((1,2)) = \frac{|\{T_k \in D \mid X \subseteq T_k\}|}{|D|} = 3/5$$

- That is: relation of number of events containing itemset (1,2) to number of all events in database

Confidence (certainty)

- Certainty of a rule can be measured with a threshold for confidence
- This parameter lets to measure how often an event's itemset that matches the left side of the implication in the association rule also matches for the right side
- Rules for events whose itemsets do not match sufficiently often the right side while mathching the left (defined by a threshold value) can be excluded

Confidence (certainty) (2)

- Database D consists of events T_1, T_2,… T_m, that is:
  $D = \{T_1, T_2, \ldots, T_m\}$
- Let there be a rule $X_a \rightarrow X_b$ so that itemsets X_a and X_b are subregions of event T_k, that is: $X_a \subseteq T_k \wedge X_b \subseteq T_k$
- Also let $X_a \cap X_b = \varnothing$
- The confidence can be defined as

$$conf(X_a, X_b) = \frac{sup(X_a \cup X_b)}{sup(X_a)}$$

- This relation compares number of events containing both itemsets X_a and X_b to number of events containing an itemset X_a

Confidence (certainty), example

- Let's assume D = {(1,2,3), (2,3,4), (1,2,4), (1,2,5), (1,3,5)}
- The confidence for rule *1* ➔ *2*

$$conf((1,2)) = \frac{sup(1 \cup 2)}{sup(1)} = \frac{3/5}{4/5} = 3/4$$

- That is: relation of number of events containing both itemsets X_a and X_b to number of events containing an itemset X_a

Support and confidence

- If confidence gets a value of 100 % the rule is an **exact rule**
- Even if confidence reaches high values the rule is not useful unless the support value is high as well
- Rules that have both high confidence and support are called **strong rules**
- Some competing alternative approaches (other that Apriori) can generate useful rules even with low support values

Generating association rules

- Usually consists of two subproblems (Han and Kamber, 2001):
  1) Finding frequent itemsets whose occurences exceed a predefined minimum support threshold
  2) Deriving association rules from those frequent itemsets (with the constrains of minimum confidence threshold)
- These two subproblems are soleved iteratively until new rules no more emerge
- The second subproblem is quite straight- forward and most of the research focus is on the first subproblem

Use of Apriori algorithm

- Initial information: transactional database D and user-defined numeric minimun support threshold *min_sup*
- Algortihm uses knowledge from previous iteration phase to produce frequent itemsets
- This is reflected in the Latin origin of the name that means ”from what comes before”

Creating frequent sets

- Let's define:

  C_k as a candidate itemset of size k

  L_k as a frequent itemset of size k
- Main steps of iteration are:
  1) Find frequent set L_{k-1}
  2) Join step: C_k is generated by joining L_{k-1} with itself (cartesian product L_{k-1} x L_{k-1})
  3) Prune step (apriori property): Any (k − 1) size itemset that is not frequent cannot be a subset of a frequent k size itemset, hence should be removed
  4) Frequent set L_k has been achieved

Creating frequent sets (2)

- Algorithm uses breadth-first search and a hash tree structure to make candidate itemsets efficiently
- Then occurance frequency for each candidate itemset is counted
- Those candidate itemsets that have higher frequency than minimum support threshold are qualified to be frequent itemsets

Apriori algorithm in pseudocode

L_1= {frequent items};
for (k= 2; L_{k-1} !=∅; k++) **do begin**
    C_k= candidates generated from L_{k-1} (that is: cartesian product L_{k-1} x L_{k-1} and eliminating any k-1 size itemset that is not frequent);
    for each transaction t in database **do**
        increment the count of all candidates in C_k that are contained in t
    L_k = candidates in C_k with *min_sup*
    end
return $\cup_k L_k$;

(www.cs.sunysb.edu/~cse634/lecture_notes/07**apriori**.pdf)

Apriori algorithm in pseudocode (2)

Apriori Pseudocode

Apriori (T, ε)

$L_1 \leftarrow \{$ *large 1-itemsets that appear in more than* ε *transactions* $\}$

$k \leftarrow 2$

while $L_{k-1} \neq \varnothing$

$C_k \leftarrow$ *Generate*(L_{k-1})

for transactions $t \in T$

$C_t \leftarrow$ Subset(C_k, t)

for candidates $c \in C_t$

$\text{count}[c] \leftarrow \text{count}[c] + 1$

$L_k \leftarrow \{c \in C_k | \ \text{count}[c] \geq \varepsilon\}$

$k \leftarrow k + 1$

return $\bigcup L_k$

← Join step and prune step

(Wikipedia)

Lecture 6

Artificial Neural Networks

1 Artificial Neural Networks

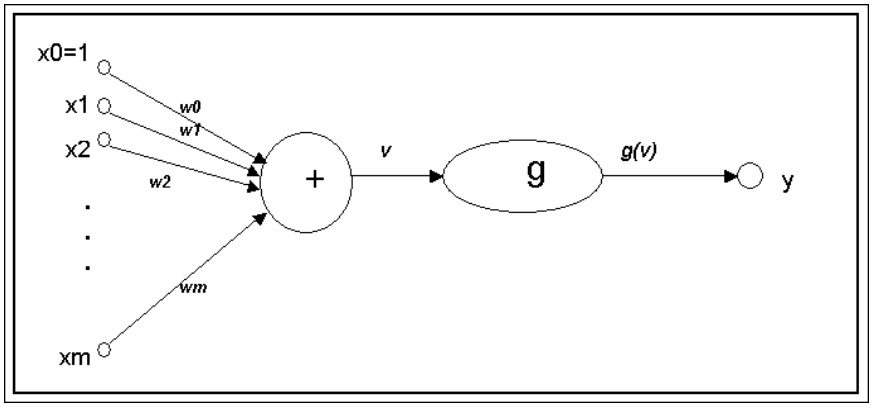
In this note we provide an overview of the key concepts that have led to the emergence of Artificial Neural Networks as a major paradigm for Data Mining applications. Neural nets have gone through two major development periods -the early 60's and the mid 80's. They were a key development in the field of machine learning. Artificial Neural Networks were inspired by biological findings relating to the behavior of the brain as a network of units called neurons. The human brain is estimated to have around 10 billion neurons each connected on average to 10,000 other neurons. Each neuron receives signals through synapses that control the effects of the signal on the neuron. These synaptic connections are believed to play a key role in the behavior of the brain. The fundamental building block in an Artificial Neural Network is the mathematical model of a neuron as shown in Figure 1. The three basic components of the (artificial) neuron are:

1. The synapses or connecting links that provide weights, w_j, to the input values, $x_j \; for \; j = 1, ...m$;

2. An adder that sums the weighted input values to compute the input to the activation function $v = w_0 + \sum_{j=1}^{m} w_j x_j$,where w_0 is called the bias (not to be confused with statistical bias in prediction or estimation) is a numerical value associated with the neuron. It is convenient to think of the bias as the weight for an input x_0 whose value is always equal to one, so that $v = \sum_{j=0}^{m} w_j x_j$;

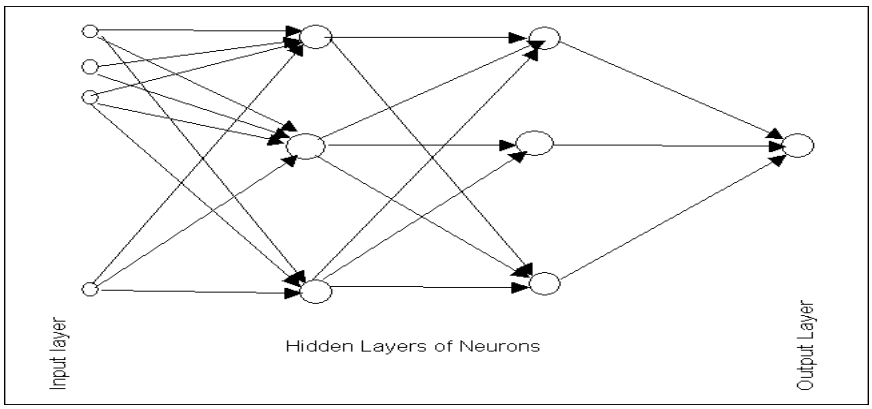
3. An activation function g (also called a squashing function) that maps v to $g(v)$ the output value of the neuron. This function is a monotone function.

Figure 1



While there are numerous different (artificial) neural network architectures that have been studied by researchers, the most successful applications in data mining of neural networks have been multilayer feedforward networks. These are networks in which there is an input layer consisting of nodes that simply accept the input values and successive layers of nodes that are neurons as depicted in Figure 1. The outputs of neurons in a layer are inputs to neurons in the next layer. The last layer is called the output layer. Layers between the input and output layers are known as hidden layers. Figure 2 is a diagram for this architecture.

Figure 2



In a supervised setting where a neural net is used to predict a numerical quantity there is one neuron in the output layer and its output is the prediction. When the network is used for classification, the output layer typically has as many nodes as the number of classes and the output layer node with

the largest output value gives the network's estimate of the class for a given input. In the special case of two classes it is common to have just one node in the output layer, the classification between the two classes being made by applying a cut-off to the output value at the node.

1.1 Single layer networks

Let us begin by examining neural networks with just one layer of neurons (output layer only, no hidden layers). The simplest network consists of just one neuron with the function g chosen to be the identity function, $g(v) = v$ for all v. In this case notice that the output of the network is $\sum_{j=0}^{m} w_j x_j$, a linear function of the input vector x with components x_j. If we are modeling the dependent variable y using multiple linear regression, we can interpret the neural network as a structure that predicts a value $\widehat{y}$ for a given input vector x with the weights being the coefficients. If we choose these weights to minimize the mean square error using observations in a training set, these weights would simply be the least squares estimates of the coefficients. The weights in neural nets are also often designed to minimize mean square error in a training data set. There is, however, a different orientation in the case of neural nets: the weights are "learned". The network is presented with cases from the training data one at a time and the weights are revised after each case in an attempt to minimize the mean square error. This process of incremental adjustment of weights is based on the error made on training cases and is known as 'training' the neural net. The almost universally used dynamic updating algorithm for the neural net version of linear regression is known as the Widrow-Hoff rule or the least-mean-square (LMS) algorithm. It is simply stated. Let $x(i)$ denote the input vector x for the i^{th} case used to train the network, and the weights *before* this case is presented to the net by the vector $w(i)$. The updating rule is $w(i+1) = w(i)+\eta(y(i)-\widehat{y}(i))x(i)$ with $w(0) = 0$. It can be shown that if the network is trained in this manner by repeatedly presenting test data observations one-at-a-time then for suitably small (absolute) values of η the network will learn (converge to) the optimal values of w. Note that the training data may have to be presented several times for $w(i)$ to be close to the optimal w. The advantage of dynamic updating is that the network tracks moderate time trends in the underlying linear model quite effectively.

If we consider using the single layer neural net for classification into c classes, we would use c nodes in the output layer. If we think of classical

discriminant analysis in neural network terms, the coefficients in Fisher's classification functions give us weights for the network that are optimal if the input vectors come from Multivariate Normal distributions with a common covariance matrix.

For classification into two classes, the linear optimization approach that we examined in class, can be viewed as choosing optimal weights in a single layer neural network using the appropriate objective function.

Maximum likelihood coefficients for logistic regression can also be considered as weights in a neural network to minimize a function of the residuals called the deviance. In this case the logistic function $g(v) = \left(\frac{e^v}{1+e^v}\right)$ is the activation function for the output node.

1.2 Multilayer Neural networks

Multilayer neural networks are undoubtedly the most popular networks used in applications. While it is possible to consider many activation functions, in practice it has been found that the logistic (also called the sigmoid) function $g(v) = \left(\frac{e^v}{1+e^v}\right)$ as the activation function (or minor variants such as the tanh function) works best. In fact the revival of interest in neural nets was sparked by successes in training neural networks using this function in place of the historically (biologically inspired) step function (the "perceptron"}. Notice that using a linear function does not achieve anything in multilayer networks that is beyond what can be done with single layer networks with linear activation functions. The practical value of the logistic function arises from the fact that it is almost linear in the range where g is between 0.1 and 0.9 but has a squashing effect on very small or very large values of v.

In theory it is sufficient to consider networks with two layers of neurons–one hidden and one output layer–and this is certainly the case for most applications. There are, however, a number of situations where three and sometimes four and five layers have been more effective. For prediction the output node is often given a linear activation function to provide forecasts that are not limited to the zero to one range. An alternative is to scale the output to the linear part (0.1 to 0.9) of the logistic function.

Unfortunately there is no clear theory to guide us on choosing the number of nodes in each hidden layer or indeed the number of layers. The common practice is to use trial and error, although there are schemes for combining

optimization methods such as genetic algorithms with network training for these parameters.

Since trial and error is a necessary part of neural net applications it is important to have an understanding of the standard method used to train a multilayered network: backpropagation. It is no exaggeration to say that the speed of the backprop algorithm made neural nets a practical tool in the manner that the simplex method made linear optimization a practical tool. The revival of strong interest in neural nets in the mid 80s was in large measure due to the efficiency of the backprop algorithm.

1.3 Example1: Fisher's Iris data

Let us look at the Iris data that Fisher analyzed using Discriminant Analysis. Recall that the data consisted of four measurements on three types of iris flowers. There are 50 observations for each class of iris. A part of the data is reproduced below.

OBS#	SPECIES	CLASSCODE	SEPLEN	SEPW	PETLEN	PETW
1	Iris-setosa	1	5.1	3.5	1.4	0.2
2	Iris-setosa	1	4.9	3	1.4	0.2
3	Iris-setosa	1	4.7	3.2	1.3	0.2
4	Iris-setosa	1	4.6	3.1	1.5	0.2
5	Iris-setosa	1	5	3.6	1.4	0.2
6	Iris-setosa	1	5.4	3.9	1.7	0.4
7	Iris-setosa	1	4.6	3.4	1.4	0.3
8	Iris-setosa	1	5	3.4	1.5	0.2
9	Iris-setosa	1	4.4	2.9	1.4	0.2
10	Iris-setosa	1	4.9	3.1	1.5	0.1
...	...	...	...	...	...	...
51	Iris-versicolor	2	7	3.2	4.7	1.4
52	Iris-versicolor	2	6.4	3.2	4.5	1.5
53	Iris-versicolor	2	6.9	3.1	4.9	1.5
54	Iris-versicolor	2	5.5	2.3	4	1.3
55	Iris-versicolor	2	6.5	2.8	4.6	1.5
56	Iris-versicolor	2	5.7	2.8	4.5	1.3
57	Iris-versicolor	2	6.3	3.3	4.7	1.6
58	Iris-versicolor	2	4.9	2.4	3.3	1
59	Iris-versicolor	2	6.6	2.9	4.6	1.3
60	Iris-versicolor	2	5.2	2.7	3.9	1.4
...	...	...	...	...	...	...
101	Iris-virginica	3	6.3	3.3	6	2.5
102	Iris-virginica	3	5.8	2.7	5.1	1.9
103	Iris-virginica	3	7.1	3	5.9	2.1
104	Iris-virginica	3	6.3	2.9	5.6	1.8
105	Iris-virginica	3	6.5	3	5.8	2.2
106	Iris-virginica	3	7.6	3	6.6	2.1
107	Iris-virginica	3	4.9	2.5	4.5	1.7
108	Iris-virginica	3	7.3	2.9	6.3	1.8
109	Iris-virginica	3	6.7	2.5	5.8	1.8
110	Iris-virginica	3	7.2	3.6	6.1	2.5

If we use a neural net architecture for this classification problem we will need 4 nodes (not counting the bias node) one for each of the 4 independent variables in the input layer and 3 neurons (one for each class) in the output layer. Let us select one hidden layer with 25 neurons. Notice that there will be a total of 25 connections from each node in the input layer to nodes in the hidden layer. This makes a total of 4 x 25 = 100 connections between

the input layer and the hidden layer. In addition there will be a total of 3 connections from each node in the hidden layer to nodes in the output layer. This makes a total of 25 x 3 = 75 connections between the hidden layer and the output layer. Using the standard logistic activation functions, the network was trained with a run consisting of 60,000 iterations. Each iteration consists of presentation to the input layer of the independent variables in a case, followed by successive computations of the outputs of the neurons of the hidden layer and the output layer using the appropriate weights. The output values of neurons in the output layer are used to compute the error. This error is used to adjust the weights of all the connections in the network using the backward propagation ("backprop") to complete the iteration. Since the training data has 150 cases, each case was presented to the network 400 times. Another way of stating this is to say the network was trained for 400 epochs where an epoch consists of one sweep through the entire training data. The results for the last epoch of training the neural net on this data are shown below:

Iris Output 1

Classification Confusion Matrix

Desired Class	Computed Class 1	2	3	Total
1	50			50
2		49	1	50
3		1	49	50
Total	50	50	50	150

Error Report

Class	Patterns	# Errors	% Errors	StdDev
1	50	0	0.00	(0.00)
2	50	1	2.00	(1.98)
3	50	1	2.00	(1.98)
Overall	150	2	1.3	(0.92)

The classification error of 1.3% is better than the error using discriminant analysis which was 2% (See lecture note on Discriminant Analysis). Notice that had we stopped after only one pass of the data (150 iterations) the

error is much worse (75%) as shown below:

Iris Output 2

Classification Confusion Matrix

Desired Class	Computed Class 1	2	3	Total
1	10	7	2	19
2	13	1	6	20
3	12	5	4	21
Total	35	13	12	60

The classification error rate of 1.3% was obtained by careful choice of key control parameters for the training run by trial and error. If we set the control parameters to poor values we can have terrible results. To understand the parameters involved we need to understand how the backward propagation algorithm works.

1.4 The Backward Propagation Algorithm

We will discuss the backprop algorithm for classification problems. There is a minor adjustment for prediction problems where we are trying to predict a continuous numerical value. In that situation we change the activation function for output layer neurons to the identity function that has *output value=input value*. (An alternative is to rescale and recenter the logistic function to permit the outputs to be approximately linear in the range of dependent variable values).

The backprop algorithm cycles through two distinct passes, a forward pass followed by a backward pass through the layers of the network. The algorithm alternates between these passes several times as it scans the training data. Typically, the training data has to be scanned several times before the networks "learns" to make good classifications.

Forward Pass: Computation of outputs of all the neurons in the network The algorithm starts with the first hidden layer using as input values the independent variables of a case (often called an exemplar in the machine learning community) from the training data set. The neuron outputs are computed for all neurons in the first hidden layer by performing

the relevant sum and activation function evaluations. These outputs are the inputs for neurons in the second hidden layer. Again the relevant sum and activation function calculations are performed to compute the outputs of second layer neurons. This continues layer by layer until we reach the output layer and compute the outputs for this layer. These output values constitute the neural net's guess at the value of the dependent variable. If we are using the neural net for classification, and we have c classes, we will have c neuron outputs from the activation functions and we use the largest value to determine the net's classification. (If $c = 2$, we can use just one output node with a cut-off value to map an numerical output value to one of the two classes).

Let us denote by w_{ij} the weight of the connection from node i to node j. The values of w_{ij} are initialized to small (generally random) numbers in the range 0.00 ± 0.05. These weights are adjusted to new values in the backward pass as described below.

Backward pass: Propagation of error and adjustment of weights This phase begins with the computation of error at each neuron in the output layer. A popular error function is the squared difference between o_k the output of node k and y_k the target value for that node. The target value is just 1 for the output node corresponding to the class of the exemplar and zero for other output nodes.(In practice it has been found better to use values of 0.9 and 0.1 respectively.) For each output layer node compute its error term as $\delta_k = o_k(1 - o_k)(y_k - o_k)$. These errors are used to adjust the weights of the connections between the last-but-one layer of the network and the output layer. The adjustment is similar to the simple Widrow-Huff rule that we saw earlier in this note. The new value of the weight w_{jk} of the connection from node j to node k is given by: $w_{jk}^{new} = w_{jk}^{old} + \eta o_j \delta_k$. Here η is an important tuning parameter that is chosen by trial and error by repeated runs on the training data. Typical values for η are in the range 0.1 to 0.9. Low values give slow but steady learning, high values give erratic learning and may lead to an unstable network.

The process is repeated for the connections between nodes in the last hidden layer and the last-but-one hidden layer. The weight for the connection between nodes i and j is given by: $w_{ij}^{new} = w_{ij}^{old} + \eta o_i \delta_j$ where $\delta_j = o_j(1 - o_j) \sum_k w_{jk}\ \delta_k$, for each node j in the last hidden layer.

The backward propagation of weight adjustments along these lines continues until we reach the input layer. At this time we have a new set of weights on which we can make a new forward pass when presented with a training data observation.

1.4.1 Multiple Local Optima and Epochs

The backprop algorithm is a version of the steepest descent optimization method applied to the problem of finding the weights that minimize the error function of the network output. Due to the complexity of the function and the large numbers of weights that are being "trained" as the network "learns", there is no assurance that the backprop algorithm (and indeed any practical algorithm) will find the optimum weights that minimize error. the procedure can get stuck at a local minimum. It has been found useful to randomize the order of presentation of the cases in a training set between different scans. It is possible to speed up the algorithm by batching, that is updating the weights for several exemplars in a pass. However, at least the extreme case of using the entire training data set on each update has been found to get stuck frequently at poor local minima.

A single scan of all cases in the training data is called an epoch. Most applications of feedforward networks and backprop require several epochs before errors are reasonably small. A number of modifications have been proposed to reduce the epochs needed to train a neural net. One commonly employed idea is to incorporate a momentum term that injects some inertia in the weight adjustment on the backward pass. This is done by adding a term to the expression for weight adjustment for a connection that is a fraction of the previous weight adjustment for that connection. This fraction is called the momentum control parameter. High values of the momentum parameter will force successive weight adjustments to be in similar directions. Another idea is to vary the adjustment parameter δ so that it decreases as the number of epochs increases. Intuitively this is useful because it avoids overfitting that is more likely to occur at later epochs than earlier ones.

1.4.2 Overfitting and the choice of training epochs

A weakness of the neural network is that it can be easily overfitted, causing the error rate on validation data to be much larger than the error rate on the training data. It is therefore important not to overtrain the data. A good method for choosing the number of training epochs is to use the validation data set periodically to compute the error rate for it while the network is being trained. The validation error decreases in the early epochs of backprop but after a while it begins to increase. The point of minimum validation error is a good indicator of the best number of epochs for training and the weights at that stage are likely to provide the best error rate in new data.

1.5 Adaptive Selection of Architecture

One of the time consuming and complex aspects of using backprop is that we need to decide on an architecture before we can use backprop. The usual procedure is to make intelligent guesses using past experience and to do several trial and error runs on different architectures. Algorithms exist that grow the number of nodes selectively during training or trim them in a manner analogous to what we have seen with CART. Research continues on such methods. However, as of now there seems to be no automatic method that is clearly superior to the trial and error approach.

1.6 Successful Applications

There have been a number of very successful applications of neural nets in engineering applications. One of the well known ones is ALVINN that is an autonomous vehicle driving application for normal speeds on highways. The neural net uses a 30x32 grid of pixel intensities from a fixed camera on the vehicle as input, the output is the direction of steering. It uses 30 output units representing classes such as "sharp left", "straight ahead", and "bear right". It has 960 input units and a single layer of 4 hidden neurons. The backprop algorithm is used to train ALVINN.

A number of successful applications have been reported in financial applications (see reference 2) such as bankruptcy predictions, currency market trading, picking stocks and commodity trading. Credit card and CRM applications have also been reported.

2 References

1. Bishop, Christopher: Neural Networks for Pattern Recognition, Oxford, 1995.

2. Trippi, Robert and Turban, Efraim (editors): Neural Networks in Finance and Investing, McGraw Hill 1996.

Chapter 9

DECISION TREES

Lior Rokach
Department of Industrial Engineering
Tel-Aviv University
liorr@eng.tau.ac.il

Oded Maimon
Department of Industrial Engineering
Tel-Aviv University
maimon@eng.tau.ac.il

Abstract Decision Trees are considered to be one of the most popular approaches for representing classifiers. Researchers from various disciplines such as statistics, machine learning, pattern recognition, and Data Mining have dealt with the issue of growing a decision tree from available data. This paper presents an updated survey of current methods for constructing decision tree classifiers in a top-down manner. The chapter suggests a unified algorithmic framework for presenting these algorithms and describes various splitting criteria and pruning methodologies.

Keywords: Decision tree, Information Gain, Gini Index, Gain Ratio, Pruning, Minimum Description Length, C4.5, CART, Oblivious Decision Trees

1. Decision Trees

A decision tree is a classifier expressed as a recursive partition of the instance space. The decision tree consists of nodes that form a *rooted tree*, meaning it is a *directed tree* with a node called "root" that has no incoming edges. All other nodes have exactly one incoming edge. A node with outgoing edges is called an *internal* or test node. All other nodes are called leaves (also known as terminal or decision nodes). In a decision tree, each internal node splits the instance space into two or more sub-spaces according to a certain discrete function of the input attributes values. In the simplest and most fre-

quent case, each test considers a single attribute, such that the instance space is partitioned according to the attribute's value. In the case of numeric attributes, the condition refers to a range.

Each leaf is assigned to one class representing the most appropriate target value. Alternatively, the leaf may hold a probability vector indicating the probability of the target attribute having a certain value. Instances are classified by navigating them from the root of the tree down to a leaf, according to the outcome of the tests along the path. Figure 9.1 describes a decision tree that reasons whether or not a potential customer will respond to a direct mailing. Internal nodes are represented as circles, whereas leaves are denoted as triangles. Note that this decision tree incorporates both nominal and numeric attributes. Given this classifier, the analyst can predict the response of a potential customer (by sorting it down the tree), and understand the behavioral characteristics of the entire potential customers population regarding direct mailing. Each node is labeled with the attribute it tests, and its branches are labeled with its corresponding values.

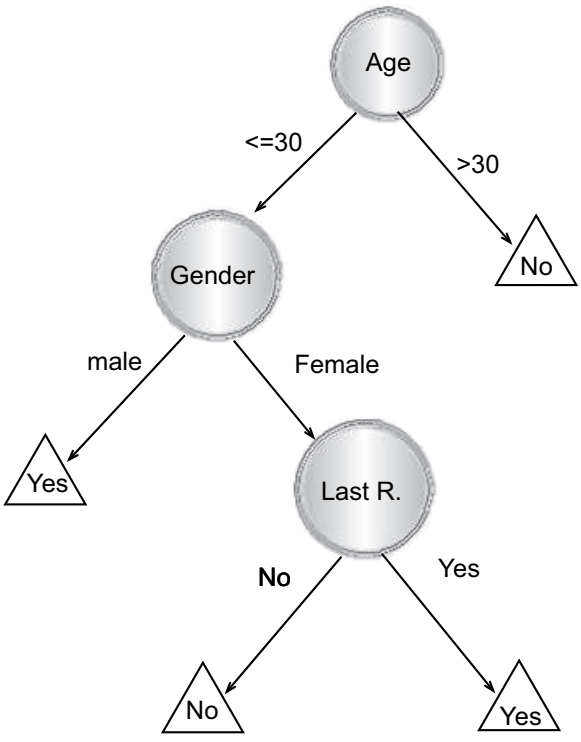


Figure 9.1. Decision Tree Presenting Response to Direct Mailing.

In case of numeric attributes, decision trees can be geometrically interpreted as a collection of hyperplanes, each orthogonal to one of the axes. Naturally, decision-makers prefer less complex decision trees, since they may be considered more comprehensible. Furthermore, according to Breiman *et al*. (1984) the tree complexity has a crucial effect on its accuracy. The tree complexity is explicitly controlled by the stopping criteria used and the pruning method employed. Usually the tree complexity is measured by one of the following metrics: the total number of nodes, total number of leaves, tree depth and number of attributes used. Decision tree induction is closely related to rule induction. Each path from the root of a decision tree to one of its leaves can be transformed into a rule simply by conjoining the tests along the path to form the antecedent part, and taking the leaf's class prediction as the class value. For example, one of the paths in Figure 9.1 can be transformed into the rule: "If customer age is is less than or equal to or equal to 30, and the gender of the customer is "Male" – then the customer will respond to the mail". The resulting rule set can then be simplified to improve its comprehensibility to a human user, and possibly its accuracy (Quinlan, 1987).

2. Algorithmic Framework for Decision Trees

Decision tree inducers are algorithms that automatically construct a decision tree from a given dataset. Typically the goal is to find the optimal decision tree by minimizing the generalization error. However, other target functions can be also defined, for instance, minimizing the number of nodes or minimizing the average depth.

Induction of an optimal decision tree from a given data is considered to be a hard task. It has been shown that finding a minimal decision tree consistent with the training set is NP–hard (Hancock *et al*., 1996). Moreover, it has been shown that constructing a minimal binary tree with respect to the expected number of tests required for classifying an unseen instance is NP–complete (Hyafil and Rivest, 1976). Even finding the minimal equivalent decision tree for a given decision tree (Zantema and Bodlaender, 2000) or building the optimal decision tree from decision tables is known to be NP–hard (Naumov, 1991).

The above results indicate that using optimal decision tree algorithms is feasible only in small problems. Consequently, heuristics methods are required for solving the problem. Roughly speaking, these methods can be divided into two groups: top–down and bottom–up with clear preference in the literature to the first group.

There are various top–down decision trees inducers such as ID3 (Quinlan, 1986), C4.5 (Quinlan, 1993), CART (Breiman *et al*., 1984). Some consist of two conceptual phases: growing and pruning (C4.5 and CART). Other inducers perform only the growing phase.

Figure 9.2 presents a typical algorithmic framework for top–down inducing of a decision tree using growing and pruning. Note that these algorithms are greedy by nature and construct the decision tree in a top–down, recursive manner (also known as "divide and conquer"). In each iteration, the algorithm considers the partition of the training set using the outcome of a discrete function of the input attributes. The selection of the most appropriate function is made according to some splitting measures. After the selection of an appropriate split, each node further subdivides the training set into smaller subsets, until no split gains sufficient splitting measure or a stopping criteria is satisfied.

3. Univariate Splitting Criteria

3.1 Overview

In most of the cases, the discrete splitting functions are univariate. Univariate means that an internal node is split according to the value of a single attribute. Consequently, the inducer searches for the best attribute upon which to split. There are various univariate criteria. These criteria can be characterized in different ways, such as:

- According to the origin of the measure: information theory, dependence, and distance.
- According to the measure structure: impurity based criteria, normalized impurity based criteria and Binary criteria.

The following section describes the most common criteria in the literature.

3.2 Impurity-based Criteria

Given a random variable x with k discrete values, distributed according to $P = (p_1, p_2, \ldots, p_k)$, an impurity measure is a function ϕ:[0, 1]$^k \rightarrow R$ that satisfies the following conditions:

- ϕ (P)$\geq$0
- ϕ (P) is minimum if $\exists$i such that component p_i = 1.
- ϕ (P) is maximum if $\forall$i, 1 $\leq$ i $\leq$ k, p_i = 1/k.
- ϕ (P) is symmetric with respect to components of P.
- ϕ (P) is smooth (differentiable everywhere) in its range.

```
TreeGrowing (S,A,y)
Where:
S - Training Set
A - Input Feature Set
y - Target Feature
Create a new tree T with a single root node.
IF One of the Stopping Criteria is fulfilled THEN
    Mark the root node in T as a leaf with the most
    common value of y in S as a label.
ELSE
    Find a discrete function f(A) of the input
        attributes values such that splitting S
        according to f(A)'s outcomes (v_1,...,v_n) gains
        the best splitting metric.
    IF best splitting metric > treshold THEN
        Label t with f(A)
        FOR each outcome v_i of f(A):
            Set Subtree_i= TreeGrowing (σ_{f(A)=v_i} S,A,y).
            Connect the root node of t_T to Subtree_i with
                    an edge that is labelled as v_i
        END FOR
    ELSE
        Mark the root node in T as a leaf with the most
            common value of y in S as a label.
    END IF
END IF
RETURN T
TreePruning (S,T,y)
Where:
S - Training Set
y - Target Feature
T - The tree to be pruned
DO
    Select a node t in T such that pruning it
        maximally improve some evaluation criteria
    IF t≠Ø THEN T=pruned(T,t)
UNTIL t=Ø
RETURN T
```

Figure 9.2. Top-Down Algorithmic Framework for Decision Trees Induction.

Note that if the probability vector has a component of 1 (the variable x gets only one value), then the variable is defined as pure. On the other hand, if all components are equal, the level of impurity reaches maximum.

Given a training set S, the probability vector of the target attribute y is defined as:

$$P_y(S) = \left(\frac{|\sigma_{y=c_1} S|}{|S|}, \ldots, \frac{\left|\sigma_{y=c_{|dom(y)|}} S\right|}{|S|} \right)$$

The goodness–of–split due to discrete attribute a_i is defined as reduction in impurity of the target attribute after partitioning S according to the values $v_{i,j} \in dom(a_i)$:

$$\Delta\Phi(a_i, S) = \phi(P_y(S)) - \sum_{j=1}^{|dom(a_i)|} \frac{|\sigma_{a_i=v_{i,j}} S|}{|S|} \cdot \phi(P_y(\sigma_{a_i=v_{i,j}} S))$$

3.3 Information Gain

Information gain is an impurity-based criterion that uses the entropy measure (origin from information theory) as the impurity measure (Quinlan, 1987).

$$InformationGain(a_i, S) = Entropy(y, S) - \sum_{v_{i,j} \in dom(a_i)} \frac{|\sigma_{a_i=v_{i,j}} S|}{|S|} \cdot Entropy(y, \sigma_{a_i=v_{i,j}} S)$$

where:

$$Entropy(y, S) = \sum_{c_j \in dom(y)} -\frac{\left|\sigma_{y=c_j} S\right|}{|S|} \cdot \log_2 \frac{\left|\sigma_{y=c_j} S\right|}{|S|}$$

3.4 Gini Index

Gini index is an impurity-based criterion that measures the divergences between the probability distributions of the target attribute's values. The Gini index has been used in various works such as (Breiman *et al*., 1984) and (Gelfand *et al*., 1991) and it is defined as:

$$Gini(y, S) = 1 - \sum_{c_j \in dom(y)} \left(\frac{\left|\sigma_{y=c_j} S\right|}{|S|} \right)^2$$

Consequently the evaluation criterion for selecting the attribute a_i is defined as:

$$GiniGain(a_i, S) = Gini(y, S) - \sum_{v_{i,j} \in dom(a_i)} \frac{\left|\sigma_{a_i=v_{i,j}} S\right|}{|S|} \cdot Gini(y, \sigma_{a_i=v_{i,j}} S)$$

3.5 Likelihood-Ratio Chi–Squared Statistics

The likelihood–ratio is defined as (Attneave, 1959)

$$G^2(a_i, S) = 2 \cdot \ln(2) \cdot |S| \cdot InformationGain(a_i, S)$$

This ratio is useful for measuring the statistical significance of the information gain criterion. The zero hypothesis (H_0) is that the input attribute and the target attribute are conditionally independent. If H_0 holds, the test statistic is distributed as χ^2 with degrees of freedom equal to: $(dom(a_i) - 1) \cdot (dom(y) - 1)$.

3.6 DKM Criterion

The DKM criterion is an impurity-based splitting criterion designed for binary class attributes (Dietterich *et al*., 1996) and (Kearns and Mansour, 1999). The impurity-based function is defined as:

$$DKM(y, S) = 2 \cdot \sqrt{\left(\frac{|\sigma_{y=c_1} S|}{|S|}\right) \cdot \left(\frac{|\sigma_{y=c_2} S|}{|S|}\right)}$$

It has been theoretically proved (Kearns and Mansour, 1999) that this criterion requires smaller trees for obtaining a certain error than other impurity based criteria (information gain and Gini index).

3.7 Normalized Impurity Based Criteria

The impurity-based criterion described above is biased towards attributes with larger domain values. Namely, it prefers input attributes with many values over attributes with less values (Quinlan, 1986). For instance, an input attribute that represents the national security number will probably get the highest information gain. However, adding this attribute to a decision tree will result in a poor generalized accuracy. For that reason, it is useful to "normalize" the impurity based measures, as described in the following sections.

3.8 Gain Ratio

The gain ratio "normalizes" the information gain as follows (Quinlan, 1993):

$$GainRatio(a_i, S) = \frac{InformationGain(a_i, S)}{Entropy(a_i, S)}$$

Note that this ratio is not defined when the denominator is zero. Also the ratio may tend to favor attributes for which the denominator is very small. Consequently, it is suggested in two stages. First the information gain is calculated for all attributes. As a consequence, taking into consideration only attributes

that have performed at least as good as the average information gain, the attribute that has obtained the best ratio gain is selected. It has been shown that the gain ratio tends to outperform simple information gain criteria, both from the accuracy aspect, as well as from classifier complexity aspects (Quinlan, 1988).

3.9 Distance Measure

The distance measure, like the gain ratio, normalizes the impurity measure. However, it suggests normalizing it in a different way (Lopez de Mantras, 1991):

$$\frac{\Delta\Phi(a_i,S)}{-\sum_{v_{i,j}\in dom(a_i)}\sum_{c_k\in dom(y)}\frac{|\sigma_{a_i=v_{i,j}\ AND\ y=c_k}S|}{|S|}\cdot\log_2\frac{|\sigma_{a_i=v_{i,j}\ AND\ y=c_k}S|}{|S|}}$$

3.10 Binary Criteria

The binary criteria are used for creating binary decision trees. These measures are based on division of the input attribute domain into two sub-domains.

Let $\beta(a_i, dom_1(a_i), dom_2(a_i), S)$ denote the binary criterion value for attribute a_i over sample S when $dom_1(a_i)$ and $dom_2(a_i)$ are its corresponding subdomains. The value obtained for the optimal division of the attribute domain into two mutually exclusive and exhaustive sub-domains is used for comparing attributes.

3.11 Twoing Criterion

The gini index may encounter problems when the domain of the target attribute is relatively wide (Breiman *et al.*, 1984). In this case it is possible to employ binary criterion called twoing criterion. This criterion is defined as:

$$twoing(a_i, dom_1(a_i), dom_2(a_i), S) = 0.25\cdot\frac{\left|\sigma_{a_i\in dom_1(a_i)}S\right|}{|S|}\cdot\frac{\left|\sigma_{a_i\in dom_2(a_i)}S\right|}{|S|}\cdot\left(\sum_{c_i\in dom(y)}\left|\frac{\left|\sigma_{a_i\in dom_1(a_i)\ AND\ y=c_i}S\right|}{\left|\sigma_{a_i\in dom_1(a_i)}S\right|}-\frac{\left|\sigma_{a_i\in dom_2(a_i)\ AND\ y=c_i}S\right|}{\left|\sigma_{a_i\in dom_2(a_i)}S\right|}\right|\right)^2$$

When the target attribute is binary, the gini and twoing criteria are equivalent. For multi–class problems, the twoing criteria prefer attributes with evenly divided splits.

3.12 Orthogonal (ORT) Criterion

The ORT criterion was presented by Fayyad and Irani (1992). This binary criterion is defined as:

$$ORT(a_i, dom_1(a_i), dom_2(a_i), S) = 1 - cos\theta(P_{y,1}, P_{y,2})$$

where $\theta(P_{y,1}, P_{y,2})$ is the angle between two vectors $P_{y,1}$ and $P_{y,2}$. These vectors represent the probability distribution of the target attribute in the partitions $\sigma_{a_i \in dom_1(a_i)} S$ and $\sigma_{a_i \in dom_2(a_i)} S$ respectively.

It has been shown that this criterion performs better than the information gain and the Gini index for specific problem constellations.

3.13 Kolmogorov–Smirnov Criterion

A binary criterion that uses Kolmogorov–Smirnov distance has been proposed in Friedman (1977) and Rounds (1980). Assuming a binary target attribute, namely $dom(y) = \{c_1, c_2\}$, the criterion is defined as:

$$KS(a_i, dom_1(a_i), dom_2(a_i), S) =$$

$$\left| \frac{\left|\sigma_{a_i \in dom_1(a_i)\ AND\ y=c_1} S\right|}{\left|\sigma_{y=c_1} S\right|} - \frac{\left|\sigma_{a_i \in dom_1(a_i)\, AND\, y=c_2} S\right|}{\left|\sigma_{y=c_2} S\right|} \right|$$

This measure was extended in (Utgoff and Clouse, 1996) to handle target attributes with multiple classes and missing data values. Their results indicate that the suggested method outperforms the gain ratio criteria.

3.14 AUC–Splitting Criteria

The idea of using the AUC metric as a splitting criterion was recently proposed in (Ferri *et al.*, 2002). The attribute that obtains the maximal area under the convex hull of the ROC curve is selected. It has been shown that the AUC–based splitting criterion outperforms other splitting criteria both with respect to classification accuracy and area under the ROC curve. It is important to note that unlike impurity criteria, this criterion does not perform a comparison between the impurity of the parent node with the weighted impurity of the children after splitting.

3.15 Other Univariate Splitting Criteria

Additional univariate splitting criteria can be found in the literature, such as permutation statistics (Li and Dubes, 1986), mean posterior improvements (Taylor and Silverman, 1993) and hypergeometric distribution measures (Martin, 1997).

3.16 Comparison of Univariate Splitting Criteria

Comparative studies of the splitting criteria described above, and others, have been conducted by several researchers during the last thirty years, such as (Baker and Jain, 1976; BenBassat, 1978; Mingers, 1989; Fayyad and Irani, 1992; Buntine and Niblett, 1992; Loh and Shih, 1997; Loh and Shih, 1999; Lim *et al.*, 2000). Most of these comparisons are based on empirical results, although there are some theoretical conclusions.

Many of the researchers point out that in most of the cases, the choice of splitting criteria will not make much difference on the tree performance. Each criterion is superior in some cases and inferior in others, as the "No–Free–Lunch" theorem suggests.

4. Multivariate Splitting Criteria

In multivariate splitting criteria, several attributes may participate in a single node split test. Obviously, finding the best multivariate criteria is more complicated than finding the best univariate split. Furthermore, although this type of criteria may dramatically improve the tree's performance, these criteria are much less popular than the univariate criteria.

Most of the multivariate splitting criteria are based on the linear combination of the input attributes. Finding the best linear combination can be performed using a greedy search (Breiman *et al.*, 1984; Murthy, 1998), linear programming (Duda and Hart, 1973; Bennett and Mangasarian, 1994), linear discriminant analysis (Duda and Hart, 1973; Friedman, 1977; Sklansky and Wassel, 1981; Lin and Fu, 1983; Loh and Vanichsetakul, 1988; John, 1996) and others (Utgoff, 1989a; Lubinsky, 1993; Sethi and Yoo, 1994).

5. Stopping Criteria

The growing phase continues until a stopping criterion is triggered. The following conditions are common stopping rules:

1. All instances in the training set belong to a single value of y.
2. The maximum tree depth has been reached.
3. The number of cases in the terminal node is less than the minimum number of cases for parent nodes.
4. If the node were split, the number of cases in one or more child nodes would be less than the minimum number of cases for child nodes.
5. The best splitting criteria is not greater than a certain threshold.

6. Pruning Methods

6.1 Overview

Employing tightly stopping criteria tends to create small and under–fitted decision trees. On the other hand, using loosely stopping criteria tends to generate large decision trees that are over–fitted to the training set. Pruning methods originally suggested in (Breiman *et al.*, 1984) were developed for solving this dilemma. According to this methodology, a loosely stopping criterion is used, letting the decision tree to overfit the training set. Then the over-fitted tree is cut back into a smaller tree by removing sub–branches that are not contributing to the generalization accuracy. It has been shown in various studies that employing pruning methods can improve the generalization performance of a decision tree, especially in noisy domains.

Another key motivation of pruning is "trading accuracy for simplicity" as presented in (Bratko and Bohanec, 1994). When the goal is to produce a sufficiently accurate compact concept description, pruning is highly useful. Within this process, the initial decision tree is seen as a completely accurate one. Thus the accuracy of a pruned decision tree indicates how close it is to the initial tree.

There are various techniques for pruning decision trees. Most of them perform top-down or bottom-up traversal of the nodes. A node is pruned if this operation improves a certain criteria. The following subsections describe the most popular techniques.

6.2 Cost–Complexity Pruning

Cost-complexity pruning (also known as weakest link pruning or error-complexity pruning) proceeds in two stages (Breiman *et al.*, 1984). In the first stage, a sequence of trees T_0, T_1, ..., T_k is built on the training data where T_0 is the original tree before pruning and T_k is the root tree.

In the second stage, one of these trees is chosen as the pruned tree, based on its generalization error estimation.

The tree T_{i+1} is obtained by replacing one or more of the sub–trees in the predecessor tree T_i with suitable leaves. The sub–trees that are pruned are those that obtain the lowest increase in apparent error rate per pruned leaf:

$$\alpha = \frac{\varepsilon(pruned(T,t),S) - \varepsilon(T,S)}{|leaves(T)| - |leaves(pruned(T,t))|}$$

where $\varepsilon(T,S)$ indicates the error rate of the tree T over the sample S and |leaves(T)| denotes the number of leaves in T. *pruned(T,t)* denotes the tree obtained by replacing the node t in T with a suitable leaf.

In the second phase the generalization error of each pruned tree T_0, T_1, ..., T_k is estimated. The best pruned tree is then selected. If the given dataset is

large enough, the authors suggest breaking it into a training set and a pruning set. The trees are constructed using the training set and evaluated on the pruning set. On the other hand, if the given dataset is not large enough, they propose to use cross–validation methodology, despite the computational complexity implications.

6.3 Reduced Error Pruning

A simple procedure for pruning decision trees, known as reduced error pruning, has been suggested by Quinlan (1987). While traversing over the internal nodes from the bottom to the top, the procedure checks for each internal node, whether replacing it with the most frequent class does not reduce the tree's accuracy. In this case, the node is pruned. The procedure continues until any further pruning would decrease the accuracy.

In order to estimate the accuracy, Quinlan (1987) proposes to use a pruning set. It can be shown that this procedure ends with the smallest accurate sub–tree with respect to a given pruning set.

6.4 Minimum Error Pruning (MEP)

The minimum error pruning has been proposed in (Olaru and Wehenkel, 2003). It performs bottom–up traversal of the internal nodes. In each node it compares the l-probability error rate estimation with and without pruning.

The l-probability error rate estimation is a correction to the simple probability estimation using frequencies. If S_t denotes the instances that have reached a leaf t, then the expected error rate in this leaf is:

$$\varepsilon'(t) = 1 - \max_{c_i \in dom(y)} \frac{|\sigma_{y=c_i} S_t| + l \cdot p_{apr}(y = c_i)}{|S_t| + l}$$

where $p_{apr}(y = c_i)$ is the *a–priori* probability of y getting the value c_i, and l denotes the weight given to the *a–priori* probability.

The error rate of an internal node is the weighted average of the error rate of its branches. The weight is determined according to the proportion of instances along each branch. The calculation is performed recursively up to the leaves.

If an internal node is pruned, then it becomes a leaf and its error rate is calculated directly using the last equation. Consequently, we can compare the error rate before and after pruning a certain internal node. If pruning this node does not increase the error rate, the pruning should be accepted.

6.5 Pessimistic Pruning

Pessimistic pruning avoids the need of pruning set or cross validation and uses the pessimistic statistical correlation test instead (Quinlan, 1993).

The basic idea is that the error ratio estimated using the training set is not reliable enough. Instead, a more realistic measure, known as the continuity correction for binomial distribution, should be used:

$$\varepsilon'(T,S) = \varepsilon(T,S) + \frac{|leaves(T)|}{2 \cdot |S|}$$

However, this correction still produces an optimistic error rate. Consequently, one should consider pruning an internal node t if its error rate is within one standard error from a reference tree, namely (Quinlan, 1993):

$$\varepsilon'(pruned(T,t),S) \leq \varepsilon'(T,S) + \sqrt{\frac{\varepsilon'(T,S) \cdot (1-\varepsilon'(T,S))}{|S|}}$$

The last condition is based on statistical confidence interval for proportions. Usually the last condition is used such that T refers to a sub–tree whose root is the internal node t and S denotes the portion of the training set that refers to the node t.

The pessimistic pruning procedure performs top–down traversing over the internal nodes. If an internal node is pruned, then all its descendants are removed from the pruning process, resulting in a relatively fast pruning.

6.6 Error–based Pruning (EBP)

Error–based pruning is an evolution of pessimistic pruning. It is implemented in the well–known C4.5 algorithm.

As in pessimistic pruning, the error rate is estimated using the upper bound of the statistical confidence interval for proportions.

$$\varepsilon_{UB}(T,S) = \varepsilon(T,S) + Z_\alpha \cdot \sqrt{\frac{\varepsilon(T,S) \cdot (1-\varepsilon(T,S))}{|S|}}$$

where $\varepsilon(T,S)$ denotes the misclassification rate of the tree T on the training set S. Z is the inverse of the standard normal cumulative distribution and α is the desired significance level.

Let *subtree(T,t)* denote the subtree rooted by the node t. Let *maxchild(T,t)* denote the most frequent child node of t (namely most of the instances in S reach this particular child) and let S_t denote all instances in S that reach the node t.

The procedure performs bottom–up traversal over all nodes and compares the following values:

1. $\varepsilon_{UB}(subtree(T,t),S_t)$
2. $\varepsilon_{UB}(pruned(subtree(T,t),t),S_t)$

3. $\varepsilon_{UB}(subtree(T, maxchild(T,t)), S_{maxchild(T,t)})$

According to the lowest value the procedure either leaves the tree as is, prune the node t, or replaces the node t with the subtree rooted by *maxchild(T,t)*.

6.7 Optimal Pruning

The issue of finding optimal pruning has been studied in (Bratko and Bohanec, 1994) and (Almuallim, 1996). The first research introduced an algorithm which guarantees optimality, known as OPT. This algorithm finds the optimal pruning based on dynamic programming, with the complexity of $\Theta(|leveas(T)|^2)$, where T is the initial decision tree. The second research introduced an improvement of OPT called OPT–2, which also performs optimal pruning using dynamic programming. However, the time and space complexities of OPT–2 are both $\Theta(|leveas(T^*)| \cdot |\text{internal}(T)|)$, where T^* is the target (pruned) decision tree and T is the initial decision tree.

Since the pruned tree is habitually much smaller than the initial tree and the number of internal nodes is smaller than the number of leaves, OPT–2 is usually more efficient than OPT in terms of computational complexity.

6.8 Minimum Description Length (MDL) Pruning

The minimum description length can be used for evaluating the generalized accuracy of a node (Rissanen, 1989; Quinlan and Rivest, 1989; Mehta *et al.*, 1995). This method measures the size of a decision tree by means of the number of bits required to encode the tree. The MDL method prefers decision trees that can be encoded with fewer bits. The cost of a split at a leaf t can be estimated as (Mehta *et al.*, 1995):

$$\text{Cost(t)} = \sum_{c_i \in dom(y)} |\sigma_{y=c_i} S_t| \cdot \ln \frac{|S_t|}{|\sigma_{y=c_i} S_t|} + \frac{|dom(y)|-1}{2} \ln \frac{|S_t|}{2} + \ln \frac{\pi^{\frac{|dom(y)|}{2}}}{\Gamma(\frac{|dom(y)|}{2})}$$

where S_t denotes the instances that have reached node t. The splitting cost of an internal node is calculated based on the cost aggregation of its children.

6.9 Other Pruning Methods

There are other pruning methods reported in the literature, such as the MML (Minimum Message Length) pruning method (Wallace and Patrick, 1993) and Critical Value Pruning (Mingers, 1989).

6.10 Comparison of Pruning Methods

Several studies aim to compare the performance of different pruning techniques (Quinlan, 1987; Mingers, 1989; Esposito *et al.*, 1997). The results indicate that some methods (such as cost–complexity pruning, reduced error pruning) tend to over–pruning, i.e. creating smaller but less accurate decision trees. Other methods (like error-based pruning, pessimistic error pruning and minimum error pruning) bias toward under–pruning. Most of the comparisons concluded that the "no free lunch" theorem applies in this case also, namely there is no pruning method that in any case outperforms other pruning methods.

7. Other Issues

7.1 Weighting Instances

Some decision trees inducers may give different treatments to different instances. This is performed by weighting the contribution of each instance in the analysis according to a provided weight (between 0 and 1).

7.2 Misclassification costs

Several decision trees inducers can be provided with numeric penalties for classifying an item into one class when it really belongs in another.

7.3 Handling Missing Values

Missing values are a common experience in real-world data sets. This situation can complicate both induction (a training set where some of its values are missing) as well as classification (a new instance that miss certain values).

This problem has been addressed by several researchers (Friedman, 1977; Breiman *et al.*, 1984; Quinlan, 1989). One can handle missing values in the training set in the following way: let $\sigma_{a_i=?}S$ indicate the subset of instances in S whose a_i values are missing. When calculating the splitting criteria using attribute a_i, simply ignore all instances their values in attribute a_i are unknown, that is, instead of using the splitting criteria $\Delta\Phi(a_i, S)$ it uses $\Delta\Phi(a_i, S - \sigma_{a_i=?}S)$.

On the other hand, in case of missing values, the splitting criteria should be reduced proportionally as nothing has been learned from these instances (Quinlan, 1989). In other words, instead of using the splitting criteria $\Delta\Phi(a_i, S)$, it uses the following correction:

$$\frac{|S - \sigma_{a_i=?}S|}{|S|}\Delta\Phi(a_i, S - \sigma_{a_i=?}S).$$

In a case where the criterion value is normalized (as in the case of gain ratio), the denominator should be calculated as if the missing values represent an

additional value in the attribute domain. For instance, the Gain Ratio with missing values should be calculated as follows:

$$GainRatio(a_i, S) = \frac{\frac{\left|S-\sigma_{a_i=?}S\right|}{|S|} InformationGain(a_i, S-\sigma_{a_i=?}S)}{-\frac{\left|\sigma_{a_i=?}S\right|}{|S|}\log(\frac{\left|\sigma_{a_i=?}S\right|}{|S|}) - \sum_{v_{i,j}\in dom(a_i)} \frac{\left|\sigma_{a_i=v_{i,j}}S\right|}{|S|}\log(\frac{\left|\sigma_{a_i=v_{i,j}}S\right|}{|S|})}$$

Once a node is split, it is required to add $\sigma_{a_i=?}S$ to each one of the outgoing edges with the following corresponding weight:

$$\left|\sigma_{a_i=v_{i,j}}S\right| / |S - \sigma_{a_i=?}S|$$

The same idea is used for classifying a new instance with missing attribute values. When an instance encounters a node where its splitting criteria can be evaluated due to a missing value, it is passed through to all outgoing edges. The predicted class will be the class with the highest probability in the weighted union of all the leaf nodes at which this instance ends up.

Another approach known as *surrogate splits* was presented by Breiman *et al*. (1984) and is implemented in the CART algorithm. The idea is to find for each split in the tree a surrogate split which uses a different input attribute and which most resembles the original split. If the value of the input attribute used in the original split is missing, then it is possible to use the surrogate split. The resemblance between two binary splits over sample S is formally defined as:

$$res(a_i, dom_1(a_i), dom_2(a_i), a_j, dom_1(a_j), dom_2(a_j), S) = \frac{\left|\sigma_{a_i\in dom_1(a_i) \;\; AND \;\; a_j\in dom_1(a_j)}S\right|}{|S|} + \frac{\left|\sigma_{a_i\in dom_2(a_i) \;\; AND \;\; a_j\in dom_2(a_j)}S\right|}{|S|}$$

When the first split refers to attribute a_i and it splits *dom*(a_i) into *dom*$_1(a_i)$ and *dom*$_2(a_i)$. The alternative split refers to attribute a_j and splits its domain to *dom*$_1(a_j)$ and *dom*$_2(a_j)$.

The missing value can be estimated based on other instances (Loh and Shih, 1997). On the learning phase, if the value of a nominal attribute a_i in tuple q is missing, then it is estimated by its mode over all instances having the same target attribute value. Formally,

$$estimate(a_i, y_q, S) = \underset{v_{i,j}\in dom(a_i)}{\text{argmax}} \left|\sigma_{a_i=v_{i,j} \;\; AND \;\; y=y_q}S\right|$$

where y_q denotes the value of the target attribute in the tuple q. If the missing attribute a_i is numeric, then instead of using mode of a_i it is more appropriate to use its mean.

8. Decision Trees Inducers

8.1 ID3

The ID3 algorithm is considered as a very simple decision tree algorithm (Quinlan, 1986). ID3 uses information gain as splitting criteria. The growing stops when all instances belong to a single value of target feature or when best information gain is not greater than zero. ID3 does not apply any pruning procedures nor does it handle numeric attributes or missing values.

8.2 C4.5

C4.5 is an evolution of ID3, presented by the same author (Quinlan, 1993). It uses gain ratio as splitting criteria. The splitting ceases when the number of instances to be split is below a certain threshold. Error–based pruning is performed after the growing phase. C4.5 can handle numeric attributes. It can induce from a training set that incorporates missing values by using corrected gain ratio criteria as presented above.

8.3 CART

CART stands for Classification and Regression Trees (Breiman *et al*., 1984). It is characterized by the fact that it constructs binary trees, namely each internal node has exactly two outgoing edges. The splits are selected using the twoing criteria and the obtained tree is pruned by cost–complexity Pruning. When provided, CART can consider misclassification costs in the tree induction. It also enables users to provide prior probability distribution.

An important feature of CART is its ability to generate regression trees. Regression trees are trees where their leaves predict a real number and not a class. In case of regression, CART looks for splits that minimize the prediction squared error (the least–squared deviation). The prediction in each leaf is based on the weighted mean for node.

8.4 CHAID

Starting from the early seventies, researchers in applied statistics developed procedures for generating decision trees, such as: AID (Sonquist *et al*., 1971), MAID (Gillo, 1972), THAID (Morgan and Messenger, 1973) and CHAID (Kass, 1980). CHAID (Chisquare–Automatic–Interaction–Detection) was originally designed to handle nominal attributes only. For each input attribute a_i, CHAID finds the pair of values in V_i that is least significantly different with respect to the target attribute. The significant difference is measured by the p value obtained from a statistical test. The statistical test used depends on the type of target attribute. If the target attribute is continuous, an F test is

used. If it is nominal, then a Pearson chi–squared test is used. If it is ordinal, then a likelihood–ratio test is used.

For each selected pair, CHAID checks if the p value obtained is greater than a certain merge threshold. If the answer is positive, it merges the values and searches for an additional potential pair to be merged. The process is repeated until no significant pairs are found.

The best input attribute to be used for splitting the current node is then selected, such that each child node is made of a group of homogeneous values of the selected attribute. Note that no split is performed if the adjusted p value of the best input attribute is not less than a certain split threshold. This procedure also stops when one of the following conditions is fulfilled:

1. Maximum tree depth is reached.

2. Minimum number of cases in node for being a parent is reached, so it can not be split any further.

3. Minimum number of cases in node for being a child node is reached.

CHAID handles missing values by treating them all as a single valid category. CHAID does not perform pruning.

8.5 QUEST

The QUEST (Quick, Unbiased, Efficient, Statistical Tree) algorithm supports univariate and linear combination splits (Loh and Shih, 1997). For each split, the association between each input attribute and the target attribute is computed using the ANOVA F–test or Levene's test (for ordinal and continuous attributes) or Pearson's chi–square (for nominal attributes). If the target attribute is multinomial, two–means clustering is used to create two super–classes. The attribute that obtains the highest association with the target attribute is selected for splitting. Quadratic Discriminant Analysis (QDA) is applied to find the optimal splitting point for the input attribute. QUEST has negligible bias and it yields binary decision trees. Ten–fold cross–validation is used to prune the trees.

8.6 Reference to Other Algorithms

Table 9.1 describes other decision trees algorithms available in the literature. Obviously there are many other algorithms which are not included in this table. Nevertheless, most of these algorithms are a variation of the algorithmic framework presented above. A profound comparison of the above algorithms and many others has been conducted in (Lim *et al*., 2000).

Table 9.1. Additional Decision Tree Inducers.

Algorithm	Description	Reference
CAL5	Designed specifically for numerical–valued attributes	Muller and Wysotzki (1994)
FACT	An earlier version of QUEST. Uses statistical tests to select an attribute for splitting each node and then uses discriminant analysis to find the split point.	Loh and Vanichsetakul (1988)
LMDT	Constructs a decision tree based on multivariate tests are linear combinations of the attributes.	Brodley and Utgoff (1995)
T1	A one–level decision tree that classifies instances using only one attribute. Missing values are treated as a "special value". Support both continuous an nominal attributes.	Holte (1993)
PUBLIC	Integrates the growing and pruning by using MDL cost in order to reduce the computational complexity.	Rastogi and Shim (2000)
MARS	A multiple regression function is approximated using linear splines and their tensor products.	Friedman (1991)

9. Advantages and Disadvantages of Decision Trees

Several advantages of the decision tree as a classification tool have been pointed out in the literature:

1. Decision trees are self–explanatory and when compacted they are also easy to follow. In other words if the decision tree has a reasonable number of leaves, it can be grasped by non–professional users. Furthermore decision trees can be converted to a set of rules. Thus, this representation is considered as comprehensible.
2. Decision trees can handle both nominal and numeric input attributes.
3. Decision tree representation is rich enough to represent any discrete–value classifier.
4. Decision trees are capable of handling datasets that may have errors.
5. Decision trees are capable of handling datasets that may have missing values.

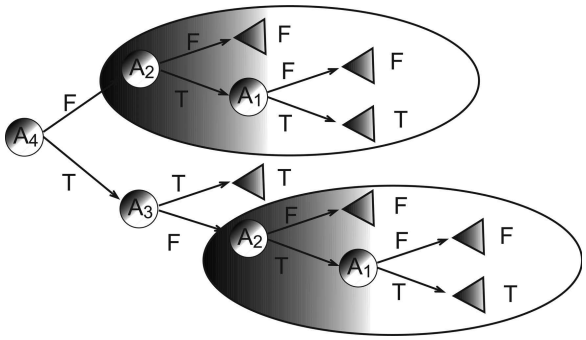


Figure 9.3. Illustration of Decision Tree with Replication.

6. Decision trees are considered to be a nonparametric method. This means that decision trees have no assumptions about the space distribution and the classifier structure.

On the other hand, decision trees have such disadvantages as:

1. Most of the algorithms (like ID3 and C4.5) require that the target attribute will have only discrete values.

2. As decision trees use the "divide and conquer" method, they tend to perform well if a few highly relevant attributes exist, but less so if many complex interactions are present. One of the reasons for this is that other classifiers can compactly describe a classifier that would be very challenging to represent using a decision tree. A simple illustration of this phenomenon is the replication problem of decision trees (Pagallo and Huassler, 1990). Since most decision trees divide the instance space into mutually exclusive regions to represent a concept, in some cases the tree should contain several duplications of the same sub-tree in order to represent the classifier. For instance if the concept follows the following binary function: $y = (A_1 \cap A_2) \cup (A_3 \cap A_4)$ then the minimal univariate decision tree that represents this function is illustrated in Figure 9.3. Note that the tree contains two copies of the same subt-ree.

3. The greedy characteristic of decision trees leads to another disadvantage that should be pointed out. This is its over–sensitivity to the training set, to irrelevant attributes and to noise (Quinlan, 1993).

10. Decision Tree Extensions

In the following sub-sections, we discuss some of the most popular extensions to the classical decision tree induction paradigm.

10.1 Oblivious Decision Trees

Oblivious decision trees are decision trees for which all nodes at the same level test the same feature. Despite its restriction, oblivious decision trees are found to be effective for feature selection. Almuallim and Dietterich (1994) as well as Schlimmer (1993) have proposed forward feature selection procedure by constructing oblivious decision trees. Langley and Sage (1994) suggested backward selection using the same means. It has been shown that oblivious decision trees can be converted to a decision table (Kohavi and Sommerfield, 1998). Recently Last *et al*. (2002) have suggested a new algorithm for constructing oblivious decision trees, called IFN (Information Fuzzy Network) that is based on information theory.

Figure 9.4 illustrates a typical oblivious decision tree with four input features: glucose level (G), age (A), hypertension (H) and pregnant (P) and the Boolean target feature representing whether that patient suffers from diabetes. Each layer is uniquely associated with an input feature by representing the interaction of that feature and the input features of the previous layers. The number that appears in the terminal nodes indicates the number of instances that fit this path. For example, regarding patients whose glucose level is less than 107 and their age is greater than 50, 10 of them are positively diagnosed with diabetes while 2 of them are not diagnosed with diabetes.

The principal difference between the oblivious decision tree and a regular decision tree structure is the constant ordering of input attributes at every terminal node of the oblivious decision tree, the property which is necessary for minimizing the overall subset of input attributes (resulting in dimensionality reduction). The arcs that connect the terminal nodes and the nodes of the target layer are labelled with the number of records that fit this path.

An oblivious decision tree is usually built by a greedy algorithm, which tries to maximize the mutual information measure in every layer. The recursive search for explaining attributes is terminated when there is no attribute that explains the target with statistical significance.

10.2 Fuzzy Decision Trees

In classical decision trees, an instance can be associated with only one branch of the tree. Fuzzy decision trees (FDT) may simultaneously assign more than one branch to the same instance with gradual certainty.

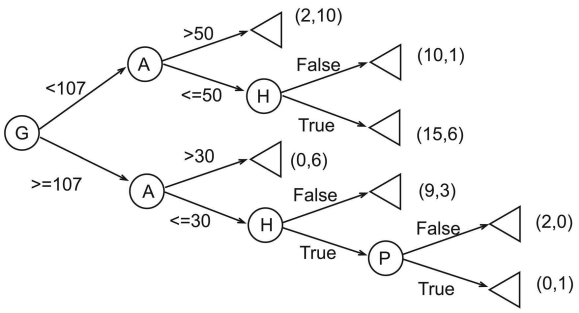


Figure 9.4. Illustration of Oblivious Decision Tree.

FDTs preserve the symbolic structure of the tree and its comprehensibility. Nevertheless, FDT can represent concepts with graduated characteristics by producing real-valued outputs with gradual shifts

Janikow (1998) presented a complete framework for building a fuzzy tree including several inference procedures based on conflict resolution in rule-based systems and efficient approximate reasoning methods.

Olaru and Wehenkel (2003) presented a new fuzzy decision trees called soft decision trees (SDT). This approach combines tree-growing and pruning, to determine the structure of the soft decision tree, with refitting and backfitting, to improve its generalization capabilities. They empirically showed that soft decision trees are significantly more accurate than standard decision trees. Moreover, a global model variance study shows a much lower variance for soft decision trees than for standard trees as a direct cause of the improved accuracy.

Peng (2004) has used FDT to improve the performance of the classical inductive learning approach in manufacturing processes. Peng (2004) proposed to use soft discretization of continuous-valued attributes. It has been shown that FDT can deal with the noise or uncertainties existing in the data collected in industrial systems.

10.3 Decision Trees Inducers for Large Datasets

With the recent growth in the amount of data collected by information systems, there is a need for decision trees that can handle large datasets. Catlett (1991) has examined two methods for efficiently growing decision trees from a large database by reducing the computation complexity required for induction. However, the Catlett method requires that all data will be loaded into the main

memory before induction. That is to say, the largest dataset that can be induced is bounded by the memory size. Fifield (1992) suggests parallel implementation of the ID3 Algorithm. However, like Catlett, it assumes that all dataset can fit in the main memory. Chan and Stolfo (1997) suggest partitioning the datasets into several disjointed datasets, so that each dataset is loaded separately into the memory and used to induce a decision tree. The decision trees are then combined to create a single classifier. However, the experimental results indicate that partition may reduce the classification performance, meaning that the classification accuracy of the combined decision trees is not as good as the accuracy of a single decision tree induced from the entire dataset.

The SLIQ algorithm (Mehta *et al.*, 1996) does not require loading the entire dataset into the main memory, instead it uses a secondary memory (disk). In other words, a certain instance is not necessarily resident in the main memory all the time. SLIQ creates a single decision tree from the entire dataset. However, this method also has an upper limit for the largest dataset that can be processed, because it uses a data structure that scales with the dataset size and this data structure must be resident in main memory all the time. The SPRINT algorithm uses a similar approach (Shafer *et al.*, 1996). This algorithm induces decision trees relatively quickly and removes all of the memory restrictions from decision tree induction. SPRINT scales any impurity based split criteria for large datasets. Gehrke *et al* (2000) introduced RainForest; a unifying framework for decision tree classifiers that are capable of scaling any specific algorithms from the literature (including C4.5, CART and CHAID). In addition to its generality, RainForest improves SPRINT by a factor of three. In contrast to SPRINT, however, RainForest requires a certain minimum amount of main memory, proportional to the set of distinct values in a column of the input relation. However, this requirement is considered modest and reasonable.

Other decision tree inducers for large datasets can be found in the literature (Alsabti *et al.*, 1998; Freitas and Lavington, 1998; Gehrke *et al.*, 1999).

10.4 Incremental Induction

Most of the decision trees inducers require rebuilding the tree from scratch for reflecting new data that has become available. Several researches have addressed the issue of updating decision trees incrementally. Utgoff (1989b, 1997) presents several methods for updating decision trees incrementally. An extension to the CART algorithm that is capable of inducing incrementally is described in (Crawford *et al.*, 2002).

References

Almuallim H., An Efficient Algorithm for Optimal Pruning of Decision Trees. Artificial Intelligence 83(2): 347-362, 1996.

Almuallim H,. and Dietterich T.G., Learning Boolean concepts in the presence of many irrelevant features. Artificial Intelligence, 69: 1-2, 279-306, 1994.

Alsabti K., Ranka S. and Singh V., CLOUDS: A Decision Tree Classifier for Large Datasets, Conference on Knowledge Discovery and Data Mining (KDD-98), August 1998.

Attneave F., Applications of Information Theory to Psychology. Holt, Rinehart and Winston, 1959.

Baker E., and Jain A. K., On feature ordering in practice and some finite sample effects. In Proceedings of the Third International Joint Conference on Pattern Recognition, pages 45-49, San Diego, CA, 1976.

BenBassat M., Myopic policies in sequential classification. IEEE Trans. on Computing, 27(2):170-174, February 1978.

Bennett X. and Mangasarian O.L., Multicategory discrimination via linear programming. Optimization Methods and Software, 3:29-39, 1994.

Bratko I., and Bohanec M., Trading accuracy for simplicity in decision trees, Machine Learning 15: 223-250, 1994.

Breiman L., Friedman J., Olshen R., and Stone C.. Classification and Regression Trees. Wadsworth Int. Group, 1984.

Brodley C. E. and Utgoff. P. E., Multivariate decision trees. Machine Learning, 19:45-77, 1995.

Buntine W., Niblett T., A Further Comparison of Splitting Rules for Decision-Tree Induction. Machine Learning, 8: 75-85, 1992.

Catlett J., Mega induction: Machine Learning on Vary Large Databases, PhD, University of Sydney, 1991.

Chan P.K. and Stolfo S.J, On the Accuracy of Meta-learning for Scalable Data Mining, J. Intelligent Information Systems, 8:5-28, 1997.

Crawford S. L., Extensions to the CART algorithm. Int. J. of ManMachine Studies, 31(2):197-217, August 1989.

Dietterich, T. G., Kearns, M., and Mansour, Y., Applying the weak learning framework to understand and improve C4.5. Proceedings of the Thirteenth International Conference on Machine Learning, pp. 96-104, San Francisco: Morgan Kaufmann, 1996.

Duda, R., and Hart, P., Pattern Classification and Scene Analysis, New-York, Wiley, 1973.

Esposito F., Malerba D. and Semeraro G., A Comparative Analysis of Methods for Pruning Decision Trees. EEE Transactions on Pattern Analysis and Machine Intelligence, 19(5):476-492, 1997.

Fayyad U., and Irani K. B., The attribute selection problem in decision tree generation. In proceedings of Tenth National Conference on Artificial Intelligence, pp. 104–110, Cambridge, MA: AAAI Press/MIT Press, 1992.

Ferri C., Flach P., and Hernández-Orallo J., Learning Decision Trees Using the Area Under the ROC Curve. In Claude Sammut and Achim Hoffmann, ed-

itors, Proceedings of the 19th International Conference on Machine Learning, pp. 139-146. Morgan Kaufmann, July 2002

Fifield D. J., Distributed Tree Construction From Large Datasets, Bachelor's Honor Thesis, Australian National University, 1992.

Freitas X., and Lavington S. H., Mining Very Large Databases With Parallel Processing, Kluwer Academic Publishers, 1998.

Friedman J. H., A recursive partitioning decision rule for nonparametric classifiers. IEEE Trans. on Comp., C26:404-408, 1977.

Friedman, J. H., "Multivariate Adaptive Regression Splines", The Annual Of Statistics, 19, 1-141, 1991.

Gehrke J., Ganti V., Ramakrishnan R., Loh W., BOAT-Optimistic Decision Tree Construction. SIGMOD Conference 1999: pp. 169-180, 1999.

Gehrke J., Ramakrishnan R., Ganti V., RainForest - A Framework for Fast Decision Tree Construction of Large Datasets,Data Mining and Knowledge Discovery, 4, 2/3) 127-162, 2000.

Gelfand S. B., Ravishankar C. S., and Delp E. J., An iterative growing and pruning algorithm for classification tree design. IEEE Transaction on Pattern Analysis and Machine Intelligence, 13(2):163-174, 1991.

Gillo M. W., MAID: A Honeywell 600 program for an automatised survey analysis. Behavioral Science 17: 251-252, 1972.

Hancock T. R., Jiang T., Li M., Tromp J., Lower Bounds on Learning Decision Lists and Trees. Information and Computation 126(2): 114-122, 1996.

Holte R. C., Very simple classification rules perform well on most commonly used datasets. Machine Learning, 11:63-90, 1993.

Hyafil L. and Rivest R.L., Constructing optimal binary decision trees is NP-complete. Information Processing Letters, 5(1):15-17, 1976

Janikow, C.Z., Fuzzy Decision Trees: Issues and Methods, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 28, Issue 1, pp. 1-14. 1998.

John G. H., Robust linear discriminant trees. In D. Fisher and H. Lenz, editors, Learning From Data: Artificial Intelligence and Statistics V, Lecture Notes in Statistics, Chapter 36, pp. 375-385. Springer-Verlag, New York, 1996.

Kass G. V., An exploratory technique for investigating large quantities of categorical data. Applied Statistics, 29(2):119-127, 1980.

Kearns M. and Mansour Y., A fast, bottom-up decision tree pruning algorithm with near-optimal generalization, in J. Shavlik, ed., 'Machine Learning: Proceedings of the Fifteenth International Conference', Morgan Kaufmann Publishers, Inc., pp. 269-277, 1998.

Kearns M. and Mansour Y., On the boosting ability of top-down decision tree learning algorithms. Journal of Computer and Systems Sciences, 58(1): 109-128, 1999.

Kohavi R. and Sommerfield D., Targeting business users with decision table classifiers, in R. Agrawal, P. Stolorz & G. Piatetsky-Shapiro, eds, 'Proceed-

ings of the Fourth International Conference on Knowledge Discovery and Data Mining', AAAI Press, pp. 249-253, 1998.

Langley, P. and Sage, S., Oblivious decision trees and abstract cases. in Working Notes of the AAAI-94 Workshop on Case-Based Reasoning, pp. 113-117, Seattle, WA: AAAI Press, 1994.

Last, M., Maimon, O. and Minkov, E., Improving Stability of Decision Trees, International Journal of Pattern Recognition and Artificial Intelligence, 16: 2,145-159, 2002.

Li X. and Dubes R. C., Tree classifier design with a Permutation statistic, Pattern Recognition 19:229-235, 1986.

Lim X., Loh W.Y., and Shih X., A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms . Machine Learning 40:203-228, 2000.

Lin Y. K. and Fu K., Automatic classification of cervical cells using a binary tree classifier. Pattern Recognition, 16(1):69-80, 1983.

Loh W.Y.,and Shih X., Split selection methods for classification trees. Statistica Sinica, 7: 815-840, 1997.

Loh W.Y. and Shih X., Families of splitting criteria for classification trees. Statistics and Computing 9:309-315, 1999.

Loh W.Y. and Vanichsetakul N., Tree-structured classification via generalized discriminant Analysis. Journal of the American Statistical Association, 83: 715-728, 1988.

Lopez de Mantras R., A distance-based attribute selection measure for decision tree induction, Machine Learning 6:81-92, 1991.

Lubinsky D., Algorithmic speedups in growing classification trees by using an additive split criterion. Proc. AI&Statistics93, pp. 435-444, 1993.

Martin J. K., An exact probability metric for decision tree splitting and stopping. An Exact Probability Metric for Decision Tree Splitting and Stopping, Machine Learning, 28, 2-3):257-291, 1997.

Mehta M., Rissanen J., Agrawal R., MDL-Based Decision Tree Pruning. KDD 1995: pp. 216-221, 1995.

Mehta M., Agrawal R. and Rissanen J., SLIQ: A fast scalable classifier for Data Mining: In Proc. If the fifth Int'l Conference on Extending Database Technology (EDBT), Avignon, France, March 1996.

Mingers J., An empirical comparison of pruning methods for decision tree induction. Machine Learning, 4(2):227-243, 1989.

Morgan J. N. and Messenger R. C., THAID: a sequential search program for the analysis of nominal scale dependent variables. Technical report, Institute for Social Research, Univ. of Michigan, Ann Arbor, MI, 1973.

Muller W., and Wysotzki F., Automatic construction of decision trees for classification. Annals of Operations Research, 52:231-247, 1994.

Murthy S. K., Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. Data Mining and Knowledge Discovery, 2(4):345-389, 1998.

Naumov G.E., NP-completeness of problems of construction of optimal decision trees. Soviet Physics: Doklady, 36(4):270-271, 1991.

Niblett T. and Bratko I., Learning Decision Rules in Noisy Domains, Proc. Expert Systems 86, Cambridge: Cambridge University Press, 1986.

Olaru C., Wehenkel L., A complete fuzzy decision tree technique, Fuzzy Sets and Systems, 138(2):221–254, 2003.

Pagallo, G. and Huassler, D., Boolean feature discovery in empirical learning, Machine Learning, 5(1): 71-99, 1990.

Peng Y., Intelligent condition monitoring using fuzzy inductive learning, Journal of Intelligent Manufacturing, 15 (3): 373-380, June 2004.

Quinlan, J.R., Induction of decision trees, Machine Learning 1, 81-106, 1986.

Quinlan, J.R., Simplifying decision trees, International Journal of Man-Machine Studies, 27, 221-234, 1987.

Quinlan, J.R., Decision Trees and Multivalued Attributes, J. Richards, ed., Machine Intelligence, V. 11, Oxford, England, Oxford Univ. Press, pp. 305-318, 1988.

Quinlan, J. R., Unknown attribute values in induction. In Segre, A. (Ed.), Proceedings of the Sixth International Machine Learning Workshop Cornell, New York. Morgan Kaufmann, 1989.

Quinlan, J. R., C4.5: Programs for Machine Learning, Morgan Kaufmann, Los Altos, 1993.

Quinlan, J. R. and Rivest, R. L., Inferring Decision Trees Using The Minimum Description Length Principle. Information and Computation, 80:227-248, 1989.

Rastogi, R., and Shim, K., PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning,Data Mining and Knowledge Discovery, 4(4):315-344, 2000.

Rissanen, J., Stochastic complexity and statistical inquiry. World Scientific, 1989.

Rounds, E., A combined non-parametric approach to feature selection and binary decision tree design, Pattern Recognition 12, 313-317, 1980.

Schlimmer, J. C. , Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In Proceedings of the 1993 International Conference on Machine Learning: pp 284-290, San Mateo, CA, Morgan Kaufmann, 1993.

Sethi, K., and Yoo, J. H., Design of multicategory, multifeature split decision trees using perceptron learning. Pattern Recognition, 27(7):939-947, 1994.

Shafer, J. C., Agrawal, R. and Mehta, M. , SPRINT: A Scalable Parallel Classifier for Data Mining, Proc. 22nd Int. Conf. Very Large Databases, T. M.

Vijayaraman and Alejandro P. Buchmann and C. Mohan and Nandlal L. Sarda (eds), 544-555, Morgan Kaufmann, 1996.

Sklansky, J. and Wassel, G. N., Pattern classifiers and trainable machines. SpringerVerlag, New York, 1981.

Sonquist, J. A., Baker E. L., and Morgan, J. N., Searching for Structure. Institute for Social Research, Univ. of Michigan, Ann Arbor, MI, 1971.

Taylor P. C., and Silverman, B. W., Block diagrams and splitting criteria for classification trees. Statistics and Computing, 3(4):147-161, 1993.

Utgoff, P. E., Perceptron trees: A case study in hybrid concept representations. Connection Science, 1(4):377-391, 1989.

Utgoff, P. E., Incremental induction of decision trees. Machine Learning, 4: 161-186, 1989.

Utgoff, P. E., Decision tree induction based on efficient tree restructuring, Machine Learning 29, 1):5-44, 1997.

Utgoff, P. E., and Clouse, J. A., A Kolmogorov-Smirnoff Metric for Decision Tree Induction, Technical Report 96-3, University of Massachusetts, Department of Computer Science, Amherst, MA, 1996.

Wallace, C. S., and Patrick J., Coding decision trees, Machine Learning 11: 7-22, 1993.

Zantema, H., and Bodlaender H. L., Finding Small Equivalent Decision Trees is Hard, International Journal of Foundations of Computer Science, 11(2): 343-354, 2000.

CS229 Lecture notes

Andrew Ng

The k-means clustering algorithm

In the clustering problem, we are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$, and want to group the data into a few cohesive "clusters." Here, $x^{(i)} \in \mathbb{R}^n$ as usual; but no labels $y^{(i)}$ are given. So, this is an unsupervised learning problem.

The k-means clustering algorithm is as follows:

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.

2. Repeat until convergence: {

   For every i, set
   $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

   For each j, set
   $$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

   }

In the algorithm above, k (a parameter of the algorithm) is the number of clusters we want to find; and the cluster centroids μ_j represent our current guesses for the positions of the centers of the clusters. To initialize the cluster centroids (in step 1 of the algorithm above), we could choose k training examples randomly, and set the cluster centroids to be equal to the values of these k examples. (Other initialization methods are also possible.)

The inner-loop of the algorithm repeatedly carries out two steps: (i) "Assigning" each training example $x^{(i)}$ to the closest cluster centroid μ_j, and (ii) Moving each cluster centroid μ_j to the mean of the points assigned to it. Figure 1 shows an illustration of running k-means.

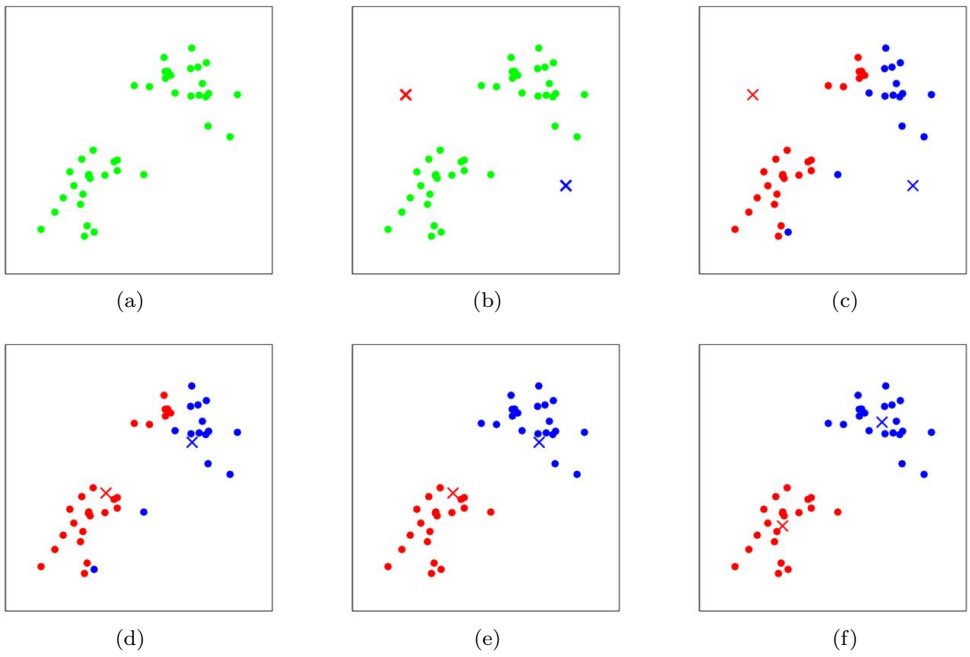


Figure 1: K-means algorithm. Training examples are shown as dots, and cluster centroids are shown as crosses. (a) Original dataset. (b) Random initial cluster centroids (in this instance, not chosen to be equal to two training examples). (c-f) Illustration of running two iterations of k-means. In each iteration, we assign each training example to the closest cluster centroid (shown by "painting" the training examples the same color as the cluster centroid to which is assigned); then we move each cluster centroid to the mean of the points assigned to it. (Best viewed in color.) Images courtesy Michael Jordan.

Is the k-means algorithm guaranteed to converge? Yes it is, in a certain sense. In particular, let us define the **distortion function** to be:

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Thus, J measures the sum of squared distances between each training example $x^{(i)}$ and the cluster centroid $\mu_{c^{(i)}}$ to which it has been assigned. It can be shown that k-means is exactly coordinate descent on J. Specifically, the inner-loop of k-means repeatedly minimizes J with respect to c while holding μ fixed, and then minimizes J with respect to μ while holding c fixed. Thus, J must monotonically decrease, and the value of J must converge. (Usually, this implies that c and μ will converge too. In theory, it is possible for

k-means to oscillate between a few different clusterings—i.e., a few different values for c and/or μ—that have exactly the same value of J, but this almost never happens in practice.)

The distortion function J is a non-convex function, and so coordinate descent on J is not guaranteed to converge to the global minimum. In other words, k-means can be susceptible to local optima. Very often k-means will work fine and come up with very good clusterings despite this. But if you are worried about getting stuck in bad local minima, one common thing to do is run k-means many times (using different random initial values for the cluster centroids μ_j). Then, out of all the different clusterings found, pick the one that gives the lowest distortion $J(c, \mu)$.

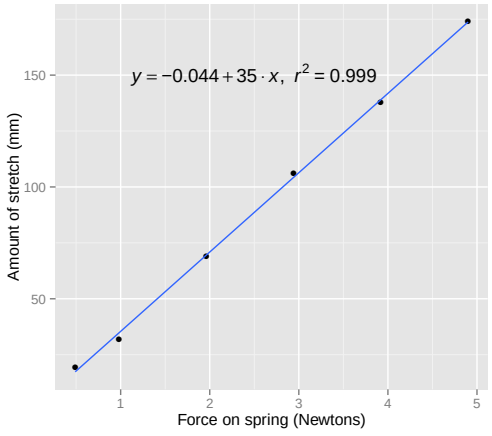
Chapter 3

Linear Regression

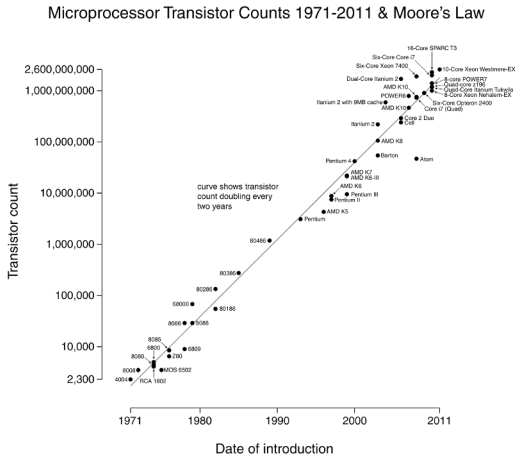
Once we've acquired data with multiple variables, one very important question is how the variables are related. For example, we could ask for the relationship between people's weights and heights, or study time and test scores, or two animal populations. **Regression** is a set of techniques for estimating relationships, and we'll focus on them for the next two chapters.

In this chapter, we'll focus on finding one of the simplest type of relationship: linear. This process is unsurprisingly called **linear regression**, and it has many applications. For example, we can relate the force for stretching a spring and the distance that the spring stretches (Hooke's law, shown in Figure 3.1a), or explain how many transistors the semiconductor industry can pack into a circuit over time (Moore's law, shown in Figure 3.1b).

Despite its simplicity, linear regression is an incredibly powerful tool for analyzing data. While we'll focus on the basics in this chapter, the next chapter will show how just a few small tweaks and extensions can enable more complex analyses.



(a) In classical mechanics, one could empirically verify Hooke's law by dangling a mass with a spring and seeing how much the spring is stretched.



(b) In the semiconductor industry, Moore's law is an observation that the number of transistors on an integrated circuit doubles roughly every two years.

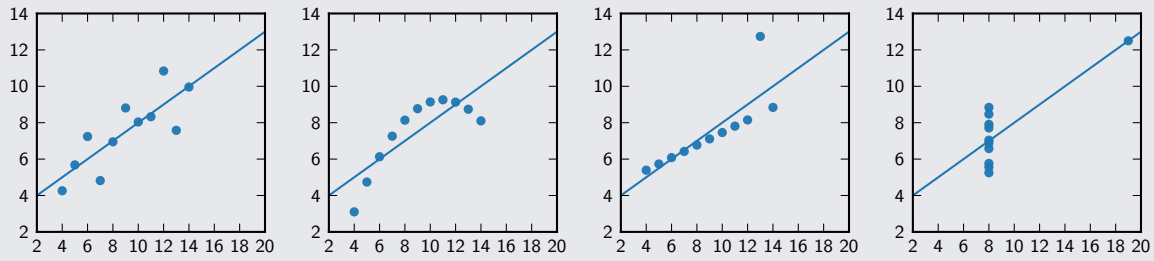
Figure 3.1: Examples of where a line fit explains physical phenomena and engineering feats.[1]

[1]The Moore's law image is by Wgsimon (own work) [CC-BY-SA-3.0 or GFDL], via Wikimedia Commons.

But just because fitting a line is easy doesn't mean that it always makes sense. Let's take another look at Anscombe's quartet to underscore this point.

Example: Anscombe's Quartet Revisited

Recall Anscombe's Quartet: 4 datasets with very similar statistical properties under a simple quantitative analysis, but that look very different. Here they are again, but this time with linear regression lines fitted to each one:



For all 4 of them, the slope of the regression line is 0.500 (to three decimal places) and the intercept is 3.00 (to two decimal places). This just goes to show: visualizing data can often reveal patterns that are hidden by pure numeric analysis!

We begin with **simple linear regression** in which there are only two variables of interest (e.g., weight and height, or force used and distance stretched). After developing intuition for this setting, we'll then turn our attention to **multiple linear regression**, where there are more variables.

Disclaimer: While some of the equations in this chapter might be a little intimidating, it's important to keep in mind that as a user of statistics, the most important thing is to understand their uses and limitations. Toward this end, make sure not to get bogged down in the details of the equations, but instead focus on understanding how they fit in to the big picture.

3.1 Simple linear regression

We're going to fit a line $y = \beta_0 + \beta_1 x$ to our data. Here, x is called the **independent variable** or **predictor variable**, and y is called the **dependent variable** or **response variable**.

Before we talk about how to do the fit, let's take a closer look at the important quantities from the fit:

- β_1 is the slope of the line: this is one of the most important quantities in any linear regression analysis. A value very close to 0 indicates little to no relationship; large positive or negative values indicate large positive or negative relationships, respectively. For our Hooke's law example earlier, the slope is the spring constant[2].

[2]Since the spring constant k is defined as $F = -kx$ (where F is the force and x is the stretch), the slope in Figure 3.1a is actually the inverse of the spring constant.

- β_0 is the intercept of the line.

In order to actually fit a line, we'll start with a way to quantify how good a line is. We'll then use this to fit the "best" line we can.

One way to quantify a line's "goodness" is to propose a probabilistic model that generates data from lines. Then the "best" line is the one for which data generated from the line is "most likely". This is a commonly used technique in statistics: proposing a probabilistic model and using the probability of data to evaluate how good a particular model is. Let's make this more concrete.

A probabilistic model for linearly related data

We observe paired data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, where we assume that as a function of x_i, each y_i is generated by using some true underlying line $y = \beta_0 + \beta_1 x$ that we evaluate at x_i, and then adding some Gaussian noise. Formally,

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i. \tag{3.1}$$

Here, the noise ε_i represents the fact that our data won't fit the model perfectly. We'll model ε_i as being Gaussian: $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. Note that the intercept β_0, the slope β_1, and the noise variance σ^2 are all treated as fixed (i.e., deterministic) but unknown quantities.

Solving for the fit: least-squares regression

Assuming that this is actually how the data $(x_1, y_1), \ldots, (x_n, y_n)$ we observe are generated, then it turns out that we can find the line for which the probability of the data is highest by solving the following optimization problem[3]:

$$\min_{\beta_0, \beta_1} : \sum_{i=1}^{n} \left[y_i - (\beta_0 + \beta_1 x_i) \right]^2, \tag{3.2}$$

where $\min_{\beta_0, \beta_1}$ means "minimize over β_0 and β_1". This is known as the **least-squares linear regression problem**. Given a set of points, the solution is:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n x_i y_i - \frac{1}{n} \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{\sum_{i=1}^n x_i^2 - \frac{1}{n} (\sum_{i=1}^n x_i)^2} \tag{3.3}$$

$$= r \frac{s_y}{s_x}, \tag{3.4}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}, \tag{3.5}$$

[3] This is an important point: the assumption of Gaussian noise leads to squared error as our minimization criterion. We'll see more regression techniques later that use different distributions and therefore different cost functions.

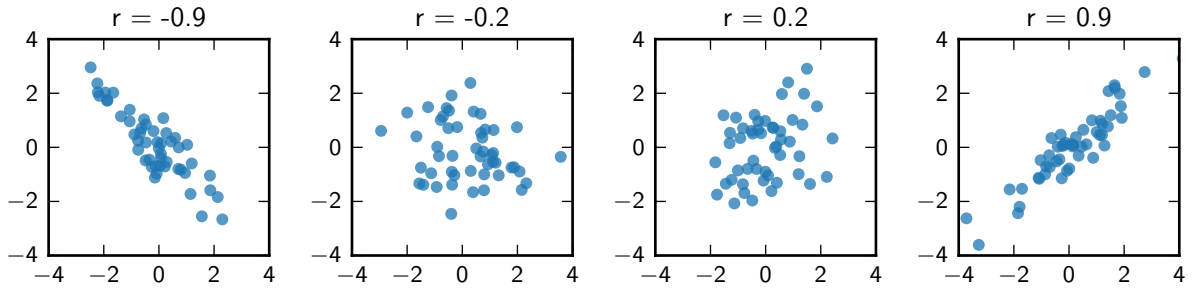


Figure 3.2: An illustration of correlation strength. Each plot shows data with a particular correlation coefficient r. Values farther than 0 (outside) indicate a stronger relationship than values closer to 0 (inside). Negative values (left) indicate an inverse relationship, while positive values (right) indicate a direct relationship.

where $\bar{x}$, $\bar{y}$, s_x and s_y are the sample means and standard deviations for x values and y values, respectively, and r is the **correlation coefficient**, defined as

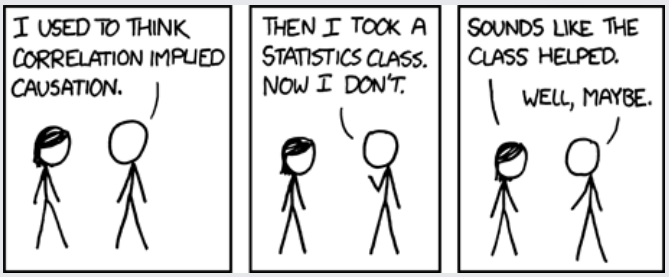
$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right). \tag{3.6}$$

By examining the second equation for the estimated slope $\hat{\beta}_1$, we see that since sample standard deviations s_x and s_y are positive quantities, the correlation coefficient r, which is always between -1 and 1, measures how much x is related to y and whether the trend is positive or negative. Figure 3.2 illustrates different correlation strengths.

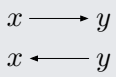
The square of the correlation coefficient r^2 will always be positive and is called the **coefficient of determination**. As we'll see later, this also is equal to the proportion of the total variability that's explained by a linear model.

As an extremely crucial remark, correlation does not imply causation! We devote the entire next page to this point, which is one of the most common sources of error in interpreting statistics.

Example: Correlation and Causation



Just because there's a strong correlation between two variables, there isn't necessarily a causal relationship between them. For example, drowning deaths and ice-cream sales are strongly correlated, but that's because both are affected by the season (summer vs. winter). In general, there are several possible cases, as illustrated below:

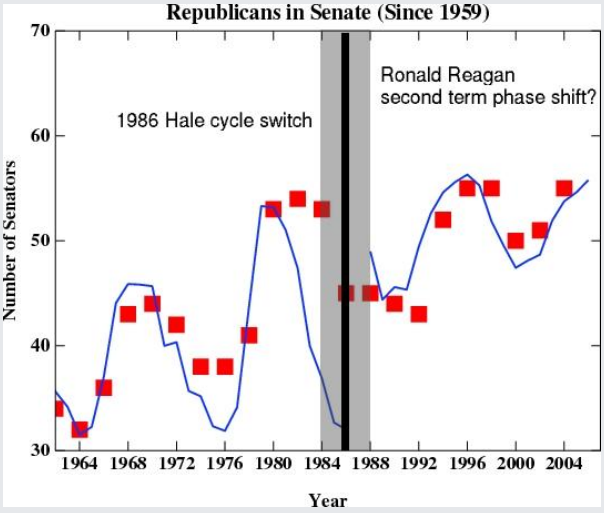


(a) **Causal link**: Even if there is a causal link between x and y, correlation alone cannot tell us whether y causes x or x causes y.

(b) **Hidden Cause**: A hidden variable z causes both x and y, creating the correlation.

(c) **Confounding Factor**: A hidden variable z and x both affect y, so the results also depend on the value of z.

(d) **Coincidence**: The correlation just happened by chance (e.g. the strong correlation between sun cycles and number of Republicans in Congress, as shown below).



(e) The number of Republican senators in congress (red) and the sunspot number (blue, before 1986)/inverted sunspot number (blue, after 1986). This figure comes from http://www.realclimate.org/index.php/archives/2007/05/fun-with-correlations/.

Figure 3.3: Different explanations for correlation between two variables. In this diagram, arrows represent causation.

3.2 Tests and Intervals

Recall from last time that in order to do hypothesis tests and compute confidence intervals, we need to know our test statistic, its standard error, and its distribution. We'll look at the standard errors for the most important quantities and their interpretation. Any statistical analysis software can compute these quantities automatically, so we'll focus on interpreting and understanding what comes out.

Warning: All the statistical tests here crucially depend on the assumption that the observed data actually comes from the probabilistic model defined in Equation (3.1)!

3.2.1 Slope

For the slope β_1, our test statistic is

$$t_{\beta_1} = \frac{\hat{\beta}_1 - \beta_1}{s_{\beta_1}}, \tag{3.7}$$

which has a Student's t distribution with $n-2$ degrees of freedom. The standard error of the slope s_{β_1} is

$$s_{\beta_1} = \frac{\hat{\sigma}}{\sqrt{\underbrace{\sum_{i=1}^{n}(x_i - \bar{x})^2}_{\text{how close together } x \text{ values are}}}} \tag{3.8}$$

and the mean squared error $\hat{\sigma}^2$ is

$$\hat{\sigma}^2 = \frac{\overbrace{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}^{\text{how large the errors are}}}{n-2} \tag{3.9}$$

These terms make intuitive sense: if the x-values are all really close together, it's harder to fit a line. This will also make our standard error s_{β_1} larger, so we'll be less confident about our slope. The standard error also gets larger as the errors grow, as we should expect it to: larger errors should indicate a worse fit.

3.2.2 Intercept

For the intercept β_0, our test statistic is

$$t_{\beta_0} = \frac{\hat{\beta}_0 - \beta_0}{s_{\beta_0}}, \tag{3.10}$$

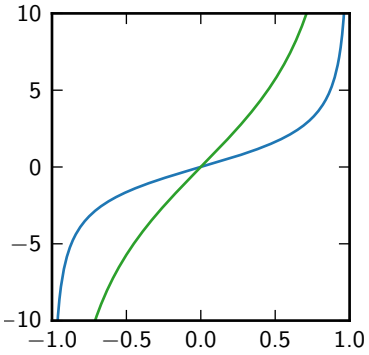


Figure 3.4: The test statistic for the correlation coefficient r for $n = 10$ (blue) and $n = 100$ (green).

which is also t-distributed with $n - 2$ degrees of freedom. The standard error is

$$s_{\beta_0} = \hat{\sigma}\sqrt{\frac{1}{n} + \frac{\bar{x}^2}{\sum_i (x_i - \bar{x})^2}}, \tag{3.11}$$

and $\hat{\sigma}$ is given by Equation (3.9).

■ 3.2.3 Correlation

For the correlation coefficient r, our test statistic is the standardized correlation

$$t_r = r\sqrt{\frac{n-2}{1-r^2}}, \tag{3.12}$$

which is t-distributed with $n - 2$ degrees of freedom. Figure 3.4 plots t_r against r.

■ 3.2.4 Prediction

Let's look at the prediction at a particular value x^*, which we'll call $\hat{y}(x^*)$. In particular:

$$\hat{y}(x^*) = \hat{\beta}_0 + \hat{\beta}_1 x^*.$$

We can do this even if x^* wasn't in our original dataset.

Let's introduce some notation that will help us distinguish between predicting the line versus predicting a particular point generated from the model. From the probabilistic model given by Equation (3.1), we can similarly write how y is generated for the new point x^*:

$$y(x^*) = \underbrace{\beta_0 + \beta_1 x^*}_{\text{defined as } \mu(x^*)} + \varepsilon, \tag{3.13}$$

where $\varepsilon \sim \mathcal{N}(0, \sigma^2)$.

Then it turns out that the standard error $s_{\hat{\mu}}$ for estimating $\mu(x^*)$ (i.e., the mean of the line at point x^*) using $\hat{y}(x^*)$ is:

$$s_{\hat{\mu}} = \hat{\sigma}\sqrt{\frac{1}{n} + \underbrace{\frac{(x^* - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}}_{\text{distance from "comfortable prediction region"}}}.$$

This makes sense because if we're trying to predict for a point that's far from the mean, then we should be less sure, and our prediction should have more variance. To compute the standard error for estimating a particular point $y(x^*)$ and not just its mean $\mu(x^*)$, we'd also need to factor in the extra noise term ε in Equation (3.13):

$$s_{\hat{y}} = \hat{\sigma}\sqrt{\frac{1}{n} + \frac{(x^* - \bar{x})}{\sum_i (x_i - \bar{x})^2} \underbrace{+1}_{\text{added}}}.$$

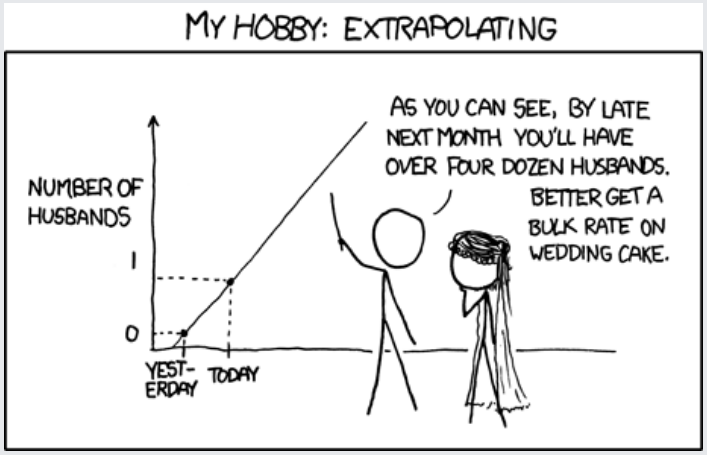
While both of these quantities have the same value when computed from the data, when analyzing them, we have to remember that they're different random variables: $\hat{y}$ has more variation because of the extra ε.

Interpolation vs. extrapolation

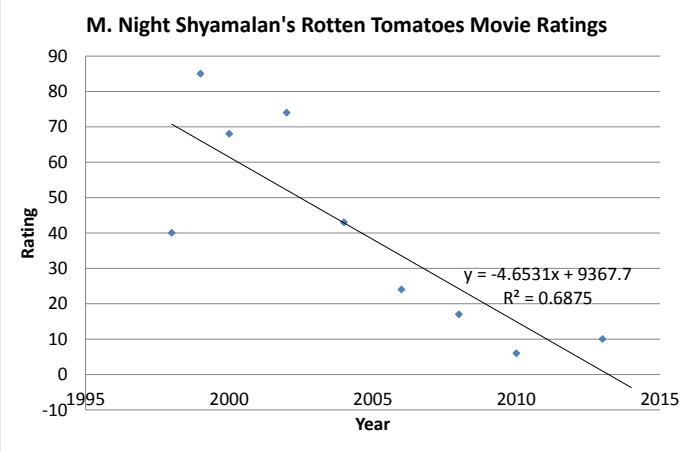
As a reminder, everything here crucially depends on the probabilistic model given by Equation (3.1) being true. In practice, when we do prediction for some value of x we haven't seen before, we need to be very careful. Predicting y for a value of x that is within the interval of points that we saw in the original data (the data that we fit our model with) is called **interpolation**. Predicting y for a value of x that's outside the range of values we actually saw for x in the original data is called **extrapolation**.

For real datasets, even if a linear fit seems appropriate, we need to be extremely careful about extrapolation, which can often lead to false predictions!

Example: The perils of extrapolation



By fitting a line to the Rotten Tomatoes ratings for movies that M. Night Shyamalan directed over time, one may erroneously be led to believe that in 2014 and onward, Shyamalan's movies will have negative ratings, which isn't even possible!



■ 3.3 Multiple Linear Regression

Now, let's talk about the case when instead of just a single scalar value x, we have a vector $(x_1, \ldots, x_p)$ for every data point i. So, we have n data points (just like before), each with p different predictor variables or **features**. We'll then try to predict y for each data point as a linear function of the different x variables:

$$y = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p. \tag{3.14}$$

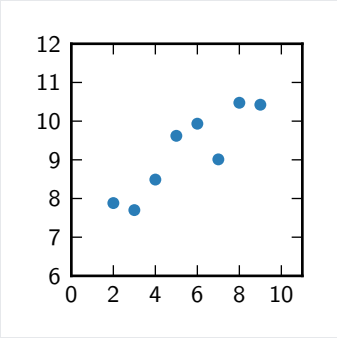
Even though it's still linear, this representation is very versatile; here are just a few of the things we can represent with it:

- Multiple dependent variables: for example, suppose we're trying to predict medical outcome as a function of several variables such as age, genetic susceptibility, and clinical diagnosis. Then we might say that for each patient, $x_1 = \text{age}, x_2 = \text{genetics}$, $x_3 = \text{diagnosis}$, and $y = \text{outcome}$.
- Nonlinearities: Suppose we want to predict a quadratic function $y = ax^2 + bx + c$: then for each data point we might say $x_1 = 1$, $x_2 = x$, and $x_3 = x^2$. This can easily be extended to any nonlinear function we want.

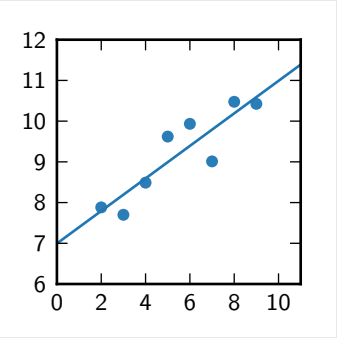
One may ask: why not just use multiple linear regression and fit an extremely high-degree polynomial to our data? While the model then would be much richer, one runs the risk of **overfitting**, where the model is so rich that it ends up fitting to the noise! We illustrate this with an example; it's also illustrated by a song[4].

Example: Overfitting

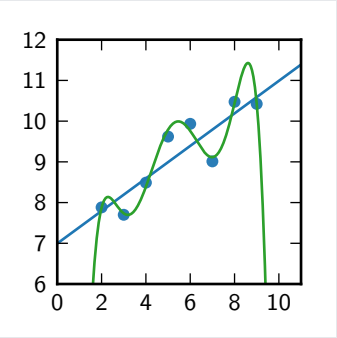
Using too many features or too complex of a model can often lead to overfitting. Suppose we want to fit a model to the points in Figure 3.3(a). If we fit a linear model, it might look like Figure 3.3(b). But, the fit isn't perfect. What if we use our newly acquired multiple regression powers to fit a 6th order polynomial to these points? The result is shown in Figure 3.3(c). While our errors are definitely smaller than they were with the linear model, the new model is far too complex, and will likely go wrong for values too far outside the range.



(a) A set of points with a simple linear relationship.



(b) The same set of points with a linear fit (blue).



(c) The same points with a 6th-order polynomial fit (green). As before, the linear fit is shown in blue.

We'll talk a little more about this in Chapters 4 and 5.

We'll represent our input data in matrix form as X, an $x \times p$ matrix where each row corresponds to a data point and each column corresponds to a feature. Since each output y_i is just a single number, we'll represent the collection as an n-element column vector y. Then our linear model can be expressed as

$$y = X\beta + \varepsilon \tag{3.15}$$

[4]Machine Learning A Cappella, Udacity. https://www.youtube.com/watch?v=DQWI1kvmwRg

where β is a p-element vector of coefficients, and ε is an n-element matrix where each element, like ε_i earlier, is normal with mean 0 and variance σ^2. Notice that in this version, we haven't explicitly written out a constant term like β_0 from before. We'll often add a column of 1s to the matrix X to accomplish this (try multiplying things out and making sure you understand why this solves the problem). The software you use might do this automatically, so it's something worth checking in the documentation.

This leads to the following optimization problem:

$$\min_{\beta} \sum_{i=1}^{n} \left(y_i - X_i\beta\right)^2, \tag{3.16}$$

where $\min_\beta$. just means "find values of β that minimize the following", and X_i refers to row i of the matrix X.

We can use some basic linear algebra to solve this problem and find the optimal estimates:

$$\hat{\beta} = (X^T X)^{-1} X^T y, \tag{3.17}$$

which most computer programs will do for you. Once we have this, what conclusions can we make with the help of statistics? We can obtain confidence intervals and/or hypothesis tests for each coefficient, which most statistical software will do for you. The test statistics are very similar to their counterparts for simple linear regression.

It's important not to blindly test whether all the coefficients are greater than zero: since this involves doing multiple comparisons, we'd need to correct appropriately using Bonferroni correction or FDR correction as described in the last chapter. But before even doing that, it's often smarter to measure whether the model even explains a significant amount of the variability in the data: if it doesn't, then it isn't even worth testing any of the coefficients individually. Typically, we'll use an **analysis of variance (ANOVA)** test to measure this. If the ANOVA test determines that the model explains a significant portion of the variability in the data, then we can consider testing each of the hypotheses and correcting for multiple comparisons.

We can also ask about which features have the most effect: if a feature's coefficient is 0 or close to 0, then that feature has little to no impact on the final result. We need to avoid the effect of scale: for example, if one feature is measured in feet and another in inches, even if they're the same, the coefficient for the feet feature will be twelve times larger. In order to avoid this problem, we'll usually look at the standardized coefficients $\frac{\hat{\beta}_k}{s_{\hat{\beta}_k}}$.

■ 3.4 Model Evaluation

How can we measure the performance of our model? Suppose for a moment that every point y_i was very close to the mean $\bar{y}$: this would mean that each y_i wouldn't depend on x_i, and that there wasn't much random error in the value either. Since we expect that this shouldn't be the case, we can try to understand how much the prediction from x_i and random error

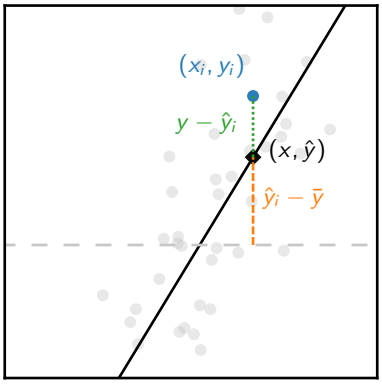


Figure 3.5: An illustration of the components contributing to the difference between the average y-value $\bar{y}$ and a particular point (x_i, y_i) (blue). Some of the difference, $\hat{y}_i - \bar{y}$, can be explained by the model (orange), and the remainder, $y_i - \hat{y}_i$, is known as the residual (green).

contribute to y_i. In particular, let's look at how far y_i is from the mean $\bar{y}$. We'll write this difference as:

$$y_i - \bar{y} = \underbrace{(\hat{y}_i - \bar{y})}_{\text{difference explained by model}} + \underbrace{(y_i - \hat{y}_i)}_{\text{difference not explained by model}} \tag{3.18}$$

In particular, the **residual** is defined to be $y_i - \hat{y}_i$: the distance from the original data point to the predicted value on the line. You can think of it as the error left over after the model has done its work. This difference is shown graphically in Figure 3.5. Note that the residual $y_i - \hat{y}$ isn't quite the same as the **noise** ε! We'll talk a little more about analyzing residuals (and why this distinction matters) in the next chapter.

If our model is doing a good job, then it should explain most of the difference from $\bar{y}$, and the first term should be bigger than the second term. If the second term is much bigger, then the model is probably not as useful.

If we square the quantity on the left, work through some algebra, and use some facts about linear regression, we'll find that

$$\underbrace{\sum_i (y_i - \bar{y})^2}_{\text{SS}_{\text{total}}} = \underbrace{\sum_i (\hat{y}_i - \bar{y})^2}_{\text{SS}_{\text{model}}} + \underbrace{\sum_i (y_i - \hat{y}_i)^2}_{\text{SS}_{\text{error}}}, \tag{3.19}$$

where "SS" stands for "sum of squares". These terms are often abbreviated as SST, SSM, and SSE respectively.

If we divide through by SST, we obtain

$$1 = \underbrace{\frac{\text{SSM}}{\text{SST}}}_{r^2} + \underbrace{\frac{\text{SSE}}{\text{SST}}}_{1-r^2},$$

where we note that r^2 is precisely the coefficient of determination mentioned earlier. Here, we see why r^2 can be interpreted as the fraction of variability in the data that is explained by the model.

One way we might evaluate a model's performance is to compare the ratio SSM/SSE. We'll do this with a slight tweak: we'll instead consider the mean values, $\text{MSM} = \text{SSM}/(p-1)$ and $\text{MSE} = \text{SSE}/(n-p)$, where the denominators correspond to the degrees of freedom. These new variables MSM and MSE have χ^2 distributions, and their ratio

$$f = \frac{\text{MSM}}{\text{MSE}} \tag{3.20}$$

has what's known as an F **distribution** with parameters $p-1$ and $n-p$. The widely used ANOVA test for categorical data, which we'll see in Chapter 6, is based on this F statistic: it's a way of measuring how much of the variability in the data is from the model and how much is from random error, and comparing the two.

LOGISTIC REGRESSION

Nitin R Patel

Logistic regression extends the ideas of multiple linear regression to the situation where the dependent variable, y, is binary (for convenience we often code these values as 0 and 1). As with multiple linear regression the independent variables $x_1, x_2 \cdots x_k$ may be categorical or continuous variables or a mixture of these two types.

Let us take some examples to illustrate [1]:

Example 1: Market Research

The data in Table 1 were obtained in a survey conducted by AT & T in the US from a national sample of co-operating households. Interest was centered on the adoption of a new telecommunications service as it related to education, residential stability and income.

Table 1: Adoption of New Telephone Service

	High School or below		Some College or above	
	No Change in Residence during Last five years	Change in Residence during Last five years	No change in Residence during Last five years	Change in Residence during Last five years
Low Income	153/2160 = 0.071	226/1137 = 0.199	61/886 = 0.069	233/1091 = 0.214
High Income	147/1363 = 0.108	139/ 547 = 0.254	287/1925 = 0.149	382/1415 = 0.270

(For fractions in cells above, the numerator is the number of adopters out of the number in the denominator).

Note that the overall probability of adoption in the sample is 1628/10524 = 0.155. However, the adoption probability varies depending on the categorical independent variables education, residential stability and income. The lowest value is 0.069 for low- income no-residence-change households with some college education while the highest is 0.270 for

high-income residence changers with some college education.

The standard multiple linear regression model is inappropriate to model this data for the following reasons:

1. The model's predicted probabilities could fall outside the range 0 to 1.

2. The dependent variable is not normally distributed. In fact a binomial model would be more appropriate. For example, if a cell total is 11 then this variable can take on only 12 distinct values $0, 1, 2 \cdots 11$. Think of the response of the households in a cell being determined by independent flips of a coin with, say, heads representing adoption with the probability of heads varying between cells.

3. If we consider the normal distribution as an approximation for the binomial model, the variance of the dependent variable is not constant across all cells: it will be higher for cells where the probability of adoption, p, is near 0.5 than where it is near 0 or 1. It will also increase with the total number of households, n, falling in the cell. The variance equals $n(p(1-p))$.

The logistic regression model was developed to account for all these difficulties. It has become very popular in describing choice behavior in econometrics and in modeling risk factors in epidemiology. In the context of choice behavior it can be shown to follow from the random utility theory developed by Manski [2] as an extension of the standard economic theory of consumer behavior.

In essence the consumer theory states that when faced with a set of choices a consumer makes a choice which has the highest utility (a numeric measure of worth with arbitrary zero and scale). It assumes that the consumer has a preference order on the list of choices that satisfies reasonable criteria such as transitivity. The preference order can depend on the individual (e.g. socioeconomic characteristics as in the Example 1 above) as well as

attributes of the choice. The random utility model considers the utility of a choice to incorporate a random element. When we model the random element as coming from a "reasonable" distribution, we can logically derive the logistic model for predicting choice behavior.

If we let $y = 1$ represent choosing an option versus $y = 0$ for not choosing it, the logistic regression model stipulates:

$$\text{Probability}(Y = 1|x_1, x_2 \cdots x_k) = \frac{\exp(\beta_O + \beta_1 * x_1 + \cdots \beta_k * x_k)}{1 + \exp(\beta_O + \beta_1 * x_1 + \cdots \beta_k * x_k)}$$

where $\beta_0, \beta_1, \beta_2 \cdots \beta_k$ are unknown constants analogous to the multiple linear regression model.

The independent variables for our model would be:

$x_1 \equiv$ (Education: High School or below = 0, Some College or above = 1

$x_2 \equiv$ (Residential Stability: No change over past five years = 0, Change over past five years = 1

$x_3 \equiv$ Income: Low = 0 High = 1

The data in Table 1 is shown below in the format typically required by regression programs.

x_1	x_2	x_3	# in sample	#adopters	# Non-adopters	Fraction adopters
0	0	0	2160	153	2007	.071
0	0	1	1363	147	1216	.108
0	1	0	1137	226	911	.199
0	1	1	547	139	408	.254
1	0	0	886	61	825	.069
1	1	0	1091	233	858	.214
1	0	1	1925	287	1638	.149
1	1	1	1415	382	1033	.270
			10524	1628	8896	1.000

The logistic model for this example is:

$$Prob(Y = 1|x_1, x_2, x_3) = \frac{\exp(\beta_0 + \beta_1 * x_l + \beta_2 * x_2 + \beta_3 * x_3)}{1 + \exp(\beta_0 + \beta_1 * x_l + \beta_2 * x_2 + \beta_3 * x_3)}.$$

We obtain a useful interpretation for the coefficients by noting that:

$$\begin{aligned}
\exp(\beta_0) &= \frac{Prob(Y = 1|x_1 = x_2 = x_3 = 0)}{Prob(Y = 0|x_1 = x_2 = x_3 = 0)} \\
&= \text{Odds of adopting in the base case } (x_1 = x_2 = x_3 = 0) \\
\exp(\beta_1) &= \frac{\text{Odds of adopting when } x_1 = 1, x_2 = x_3 = 0}{\text{Odds of adopting in the base case}} \\
\exp(\beta_2) &= \frac{\text{Odds of adopting when } x_2 = 1, x_1 = x_3 = 0}{\text{Odds of adopting in the base case}} \\
\exp(\beta_3) &= \frac{\text{Odds of adopting when } x_3 = 1, x_1 = x_2 = 0}{\text{Odds of adopting in the base case}}
\end{aligned}$$

The logistic model is multiplicative in odds in the following sense:

Odds of adopting for a given x_1, x_2, x_3

$$\begin{aligned}
&= \exp(\beta_0) * \exp(\beta_1 x_1) * \exp(\beta_2 x_2) * \exp(\beta_3 x_3) \\
&= \left\{ \begin{array}{c} Odds \\ for \\ basecase \end{array} \right\} * \left\{ \begin{array}{c} Factor \\ due \\ to\ x_1 \end{array} \right\} * \left\{ \begin{array}{c} Factor \\ due \\ to\ x_2 \end{array} \right\} * \left\{ \begin{array}{c} Factor \\ due \\ to\ x_3 \end{array} \right\}
\end{aligned}$$

If $x_1 = 1$ the odds of adoption get multiplied by the same factor for any given level of x_2 and x_3. Similarly the multiplicative factors for x_2 and x_3 do not vary with the levels of the remaining factors. The factor for a variable gives us the impact of the presence of that factor on the odds of adopting.

If $\beta_i = 0$, the presence of the corresponding factor has no effect (multiplication by one). If $\beta_i < 0$, presence of the factor reduces the odds (and the probability) of adoption, whereas if $\beta_i > 0$, presence of the factor increases the probability of adoption.

The computations required to produce these maximum likelihood estimates require iterations using a computer program. The output of a typical program is shown below:

					95% Conf. Intvl. for odds	
Variable	Coeff.	Std. Error	p-Value	Odds	Lower Limit	Upper Limit
Constant	-2.500	0.058	0.000	0.082	0.071	0.095
x_1	0.161	0.058	0.006	1.175	1.048	1.316
x_2	0.992	0.056	0.000	2.698	2.416	3.013
x_3	0.444	0.058	0.000	1.560	1.393	1.746

From the estimated values of the coefficients, we see that the estimated probability of adoption for a household with values x_1, x_2 and x_3 for the independent variables is:

$$Prob(Y = 1|x_1, x_2, x_3) = \frac{\exp(-2.500 + 0.161 * x_1 + 0.992 * x_2 + 0.444 * x_3)}{1 + \exp(-2.500 + 0.161 * x_1 + 0.992 * x_2 + 0.444 * x_3)}.$$

The estimated number of adopters from this model will be the total number of households with values x_1, x_2 and x_3 for the independent variables multiplied by the above probability.

The table below shows the estimated number of adopters for the various combinations of the independent variables.

x_1	x_2	x_3	# in sample	# adopters	Estimated (# adopters)	Fraction Adopters	Estimated $Prob(Y = l\|x_1, x_2, x_3)$
0	0	0	2160	153	164	0.071	0.076
0	0	1	1363	147	155	0.108	0.113
0	1	0	1137	226	206	0.199	0.181
0	1	1	547	139	140	0.254	0.257
1	0	0	886	61	78	0.069	0.088
1	1	0	1091	233	225	0.214	0.206
1	0	1	1925	287	252	0.149	0.131
1	1	1	1415	382	408	0.270	0.289

In data mining applications we will have validation data that is a hold-out sample not used in fitting the model.

Let us suppose we have the following validation data consisting of 598 households:

x_1	x_2	x_3	# in validation sample	# adopters in validation sample	Estimated (# adopters)	Error (Estimate -Actal)	Absolute Value of Error
0	0	0	29	3	2.200	-0.800	0.800
0	0	1	23	7	2.610	-4.390	4.390
0	1	0	112	25	20.302	-4.698	4.698
0	1	1	143	27	36.705	9.705	9.705
1	0	0	27	2	2.374	0.374	0.374
1	1	0	54	12	11.145	-0.855	0.855
1	0	1	125	13	16.338	3.338	3.338
1	1	1	85	30	24.528	-5.472	5.472
Totals			598	119	116.202		

The total error is -2.8 adopters or a percentage error in estimating adopters of -2.8/119 = 2.3%.

The average percentage absolute error is

$$\frac{0.800 + 4.390 + 4.698 + 9.705 + 0.374 + 0.855 + 3.338 + 5.472}{119}$$

$$= .249 = 24.9\% \text{ adopters.}$$

The confusion matrix for households in the validation data for set is given below:

	Observed		
	Adopters	Non-adopters	Total
Predicted:			
Adopters	103	13	116
Non-adopters	16	466	482
Total	119	479	598

As with multiple linear regression we can build more complex models that reflect interactions between independent variables by including factors that are calculated from the interacting factors. For example if we felt that there is an interactive effect b etween x_1 and x_2 we would add an interaction term $x_4 = x_1 \times x_2$.

Example 2: Financial Conditions of Banks [2]

Table 2 gives data on a sample of banks. The second column records the judgment of an expert on the financial condition of each bank. The last two columns give the values of two commonly ratios commonly used in financial analysis of banks.

Table 2: Financial Conditions of Banks

Obs	Financial Condition* (y)	Total Loans & Leases/ Total Assets (x_1)	Total Expenses / Total Assets (x_2)
1	1	0.64	0.13
2	1	1.04	0.10
3	1	0.66	0.11
4	1	0.80	0.09
5	1	0.69	0.11
6	1	0.74	0.14
7	1	0.63	0.12
8	1	0.75	0.12
9	1	0.56	0.16
10	1	0.65	0.12
11	0	0.55	0.10
12	0	0.46	0.08
13	0	0.72	0.08
14	0	0.43	0.08
15	0	0.52	0.07
16	0	0.54	0.08
17	0	0.30	0.09
18	0	0.67	0.07
19	0	0.51	0.09
20	0	0.79	0.13

$$* \text{ Financial Condition } = 1 \quad \text{for financially weak banks;}$$

$$= 0 \quad \text{for financially strong banks.}$$

Let us first consider a simple logistic regression model with just one independent variable. This is analogous to the simple linear regression model in which we fit a straight line to relate the dependent variable, y, to a single independent variable, x.

Let us construct a simple logistic regression model for classification of banks using the Total Loans & Leases to Total Assets ratio as the independent variable in our model. This model would have the following variables:

Dependent variable:

$$\begin{aligned} Y &= 1, \quad \text{if financially distressed,} \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

Independent (or Explanatory) variable:

$$x_1 = \text{Total Loans \& Leases/Total Assets Ratio}$$

The equation relating the dependent variable with the explanatory variable is:

$$Prob(Y = 1|x_1) = \frac{\exp(\beta_0 + \beta_1 * x_l)}{1 + \exp(\beta_0 + \beta_1 * x_l)}$$

or, equivalently,

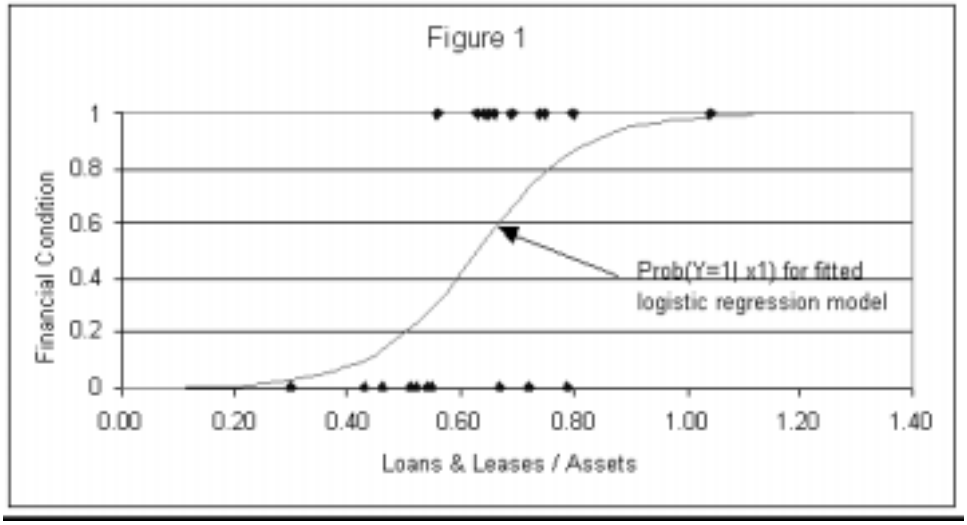
$$\text{Odds } (Y = 1 \text{ versus } Y = 0) = (\beta_0 + \beta_1 * x_l).$$

The Maximum Likelihood Estimates of the coefficients for the model are: $\hat{\beta}_0 = -6.926$, $\hat{\beta}_1 = 10.989$

So that the fitted model is:

$$Prob(Y = 1|x_1) = \frac{\exp(-6.926 + 10.989 * x_1)}{(1 + \exp(-6.926 + 10.989 * x_1)}.$$

Figure 1 displays the data points and the fitted logistic regression model.



We can think of the model as a multiplicative model of odds ratios as we did for Example 1. The odds that a bank with a Loan & Leases/Assets Ratio that is zero will be in financial distress = $\exp(-6.926) = 0.001$. These are the base case odds. The odds of distress for a bank with a ratio of 0.6 will increase by a multiplicative factor of $\exp(10.989*0.6) = 730$ over the base case, so the odds that such a bank will be in financial distress = 0.730.

Notice that there is a small difference in interpretation of the multiplicative factors for this example compared to Example 1. While the interpretation of the sign of β_i remains as before, its magnitude gives the amount by which the odds of $Y = 1$ against $Y = 0$ are changed for *a unit change* in x_i. If we construct a simple logistic regression model for classification of banks using the Total Expenses/ Total Assets ratio as the independent variable we would have the following variables:

Dependent variable:

$$\begin{aligned} Y &= 1, \quad \text{if financially distressed,} \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

Independent (or Explanatory) variables:

$$x_2 = \text{Total Expenses/ Total Assets Ratio}$$

The equation relating the dependent variable with the explanatory variable is:

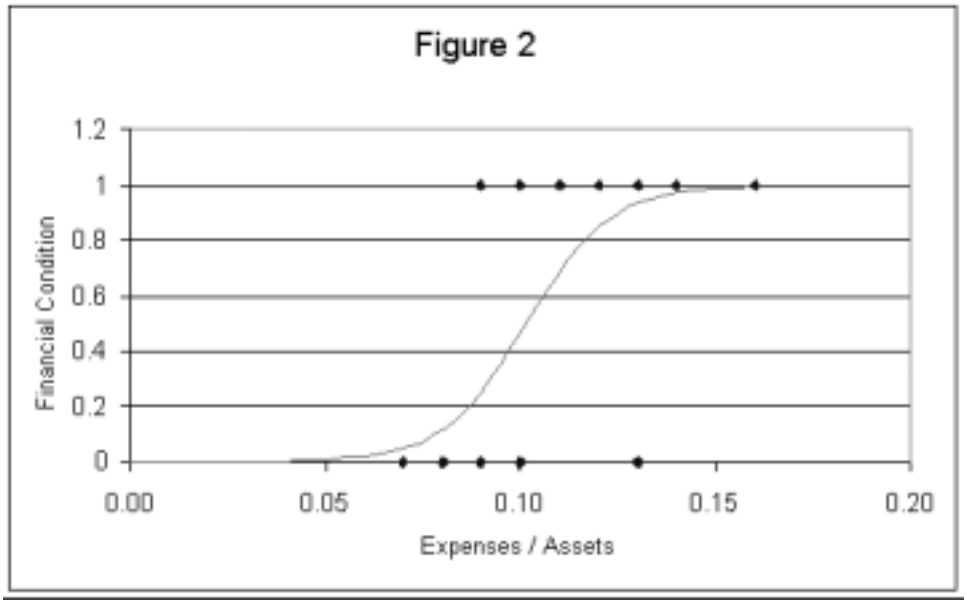
$$Prob(Y = l|x_1) = \frac{\exp(\beta_0 + \beta_2 * x_2)}{1 + \exp(\beta_0 + \beta_2 * x_2)}$$

or, equivalently,

$$\text{Odds} \quad (Y = 1 \quad \text{versus} \quad Y = 0) = (\beta_0 + \beta_2 * x_2).$$

The Maximum Likelihood Estimates of the coefficients for the model are: $\beta_0 = -9.587$, $\beta_2 = 94.345$

Figure 2 displays the data points and the fitted logistic regression model.



Computation of Estimates

As illustrated in Examples 1 and 2, estimation of coefficients is usually carried out based on the principle of *maximum likelihood* which ensures good asymptotic (large sample) properties for the estimates. Under very general conditions maximum likelihood estimators are:

- *Consistent* : the probability of the estimator differing from the true value approaches zero with increasing sample size;
- *Asymptotically Efficient* : the variance is the smallest possible among consistent estimators
- *Asymptotically Normally Distributed*: This allows us to compute confidence intervals and perform statistical tests in a manner analogous to the analysis of linear multiple regression models, provided the sample size is 'large'.

Algorithms to compute the coefficient estimates and confidence intervals are iterative and less robust than algorithms for linear regression. Computed estimates are generally reliable for well-behaved datasets where the number of observations with depende nt variable values of both 0 and 1 are 'large'; their ratio is 'not too close' to either zero or one; and when the number of coefficients in the logistic regression model is small relative to the sample size (say, no more than 10%). As with linear regression collinearity (strong correlation amongst the independent variables) can lead to computational difficulties. Computationally intensive algorithms have been developed recently that circumvent some of these difficulties [3].

Appendix A

Computing Maximum Likelihood Estimates and Confidence Intervals for Regression Coefficients

We denote the coefficients by the $p \times 1$ column vector β with the row element i equal to β_i, The n observed values of the dependent variable will be denoted by the $n \times 1$ column vector y with the row element j equal to y_j; and the corresponding values of the independent variable i by x_{ij} for

$$i = 1 \cdots p; j = 1 \cdots n.$$

Data : $y_j, x_{1j}, x_{2j}, \cdots, x_{pj}, \quad j = 1, 2, \cdots, n.$

Likelihood Function: The likelihood function, L, is the probability of the observed data viewed as a function of the parameters (β_{2i} in a logistic regression).

$$\begin{aligned}
&\prod_{j=1}^{n} \frac{e^{y_i(\beta_0+\beta_1 x_{1j}+\beta_2 x_{2j}\cdots+\beta_p x_{pj})}}{1+e^{\beta_0+\beta_1 x_{1j}+\beta_2 x_{2j}\cdots+\beta_i x_{pj})}} \\
&= \prod_{j=1}^{n} \frac{e^{\Sigma_i y_j \beta_i x_{ij}}}{1+e^{\Sigma_i \beta_i x_{ij}}} \\
&= \frac{e^{\Sigma_i(\Sigma_j y_j x_{ij})\beta_i}}{\prod_{j=1}^{n}[1+e^{\Sigma_i \beta_i x_{ij}}]} \\
&= \frac{e^{\Sigma_i \beta_i t_i}}{\prod_{j=1}^{n}[1+e^{\Sigma_i \beta_i x_{ij}}]} \\
\text{where} \quad & t_i = \Sigma_j y_j x_{ij}
\end{aligned}$$

These are the sufficient statistics for a logistic regression model analogous to $\hat{y}$ and S in linear regression.

Loglikelihood Function: This is the logarithm of the likelihood function,

$$l = \Sigma_i \beta_i t_i - \Sigma_j \log[1 + e^{\Sigma_i \beta_i x_{ij}}].$$

We find the maximum likelihood estimates, $\hat{\beta}_i$, of β_i by maximizing the loglikelihood function for the observed values of y_j and x_{ij} in our data. Since maximizing the log of a function is equivalent to maximizing the function, we often work with the loglikelihood because it is generally less cumbersome to use for mathematical operations such as differentiation.

Since the likelihood function can be shown to be concave, we will find the global maximum of the function (if it exists) by equating the partial derivatives of the loglikelihood to zero and solving the resulting nonlinear equations for $\hat{\beta}_i$.

$$\begin{aligned} \frac{\partial l}{\partial \beta_i} &= t_i - \Sigma_j \frac{x_{ij} e^{\Sigma_i \beta_i x_{ij}}}{[1 + e^{\Sigma_i \beta_i x_{ij}}]} \\ &= t_i - \Sigma_j x_{ij} \hat{\pi}_j = 0, i = 1, 2, \cdots, p \\ &\text{or} \quad \Sigma_i x_{ij} \hat{\pi}_j = t_i \end{aligned}$$

where $\hat{\pi}_j = \frac{e^{\Sigma_i b b_i x_{ij}}}{[1+e^{\Sigma_i \beta_i x_{ij}}]} = E(Y_j)$

An intuitive way to understand these equations is to note that

$$\Sigma_j x_{ij} E(Y_j) = \Sigma_j x_{ij} y_j$$

In words, the maximum likelihood estimates are such that the expected value of the sufficient statistics are equal to their observed values.

Note : If the model includes the constant term $x_{ij} = 1$ for all j then $\Sigma_j E(Y_j) = \Sigma_j y_j$, i.e. the expected number of successes (responses of one) using MLE estimates of β_i equals the observed number of successes. The $\hat{\beta}_i$'s are consistent, asymptotically efficient and follow a multivariate Normal distribution (subject to mild regularity conditions).

Algorithm : A popular algorithm for computing $\hat{\beta}_i$ uses the Newton-Raphson method for maximizing twice differentiable functions of several variables (see Appendix B).

The Newton-Raphson method involves computing the following successive approximations to find $\hat{\beta}_i$, the likelihood function

$$\beta^{t+1} = \beta^t + [I(\beta^t)]^{-1} \nabla I(\beta^t)$$

where

$$I_{ij} = \frac{\partial^2 l}{\partial \beta_i \partial_j \beta_j}$$

- On convergence, the diagonal elements of $I(\beta^t)^{-1}$ give squared standard errors (approximate variance) for $\hat{\beta}_i$.

- Confidence intervals and hypothesis tests are based on asymptotic normal distribution of $\hat{\beta}_i$.

  The loglikelihood function is always negative and does not have a maximum when it can be made arbitrary close to zero. In that case the likelihood function can be made arbitrarily close to one and the first term of the loglikelihood function given above approaches infinity. In this situation the predicted probabilities for observations with $y_j = 0$ can be made arbitrarily close to 0 and those for $y_j = 1$ can be made arbitrarily close to 1 by choosing suitable very large absolute values of some β_i. This is the situation when we have a perfect model (at least in terms of the training data set)! This phenomenon is more likely to occur when the number of parameters is a large fraction (say $> 20\%$) of the number of observations.

Appendix B

The Newton-Raphson Method

This method finds the values of β_i that maximize a twice differentiable concave function, $g(\beta)$. If the function is not concave, it finds a local maximum. The method uses successive quadratic approximations to g based on Taylor series. It converges rapidly if the starting value, β^0, is reasonably close to the maximizing value, $\hat{\beta}$, of β.

The gradient vector ∇ and the Hessian matrix, H, as defined below, are used to update an estimate β^t to β^{t+1}.

$$\nabla g(\beta^t) = \begin{bmatrix} \vdots \\ \frac{\partial g}{\partial \beta_i} \\ \vdots \end{bmatrix}_{\beta^t} \qquad H(\beta^t) = \begin{bmatrix} \cdots & \vdots & \cdots \\ & \frac{\partial^2 g}{\partial \beta_i \partial \beta_k} & \\ & \vdots & \end{bmatrix}_{\beta^t}.$$

The Taylor series expansion around β^t gives us:

$$g(\beta) \simeq g(\beta^t) + \nabla g(\beta^t)(\beta - \beta^t) + 1/2(\beta - \beta^t)' H(\beta^t)(\beta - \beta^t)$$

Provided $H(\beta^t)$ is positive definite, the maximum of this approximation occurs when its derivative is zero.

$$\nabla g(\beta^t) - H(\beta^t)(\beta - \beta^t) = 0$$

or

$$\beta = \beta^t - [H(\beta^t)]^{-1} \nabla g(\beta^t).$$

This gives us a way to compute β^{t+1}, the next value in our iterations.

$$\beta^{t+1} = \beta^t - [H(\beta^t]^{-1} \nabla g(\beta^t).$$

To use this equation H should be non-singular. This is generally not a problem although sometimes numerical difficulties can arise due to collinearity.

Near the maximum the rate of convergence is quadratic as it can be shown that

$$|\beta_i^{t+1} - \hat{\beta}_i| \leq c|\beta_i^t - \hat{\beta}_i|^2 \quad \text{for some} \quad c \geq 0 \text{ when } \beta_{\mathrm{i}}^{\mathrm{t}} \text{ is near } \hat{\beta}_{\mathrm{i}} \text{ for all i.}$$

CS229 Lecture notes

Andrew Ng

Part XI

Principal components analysis

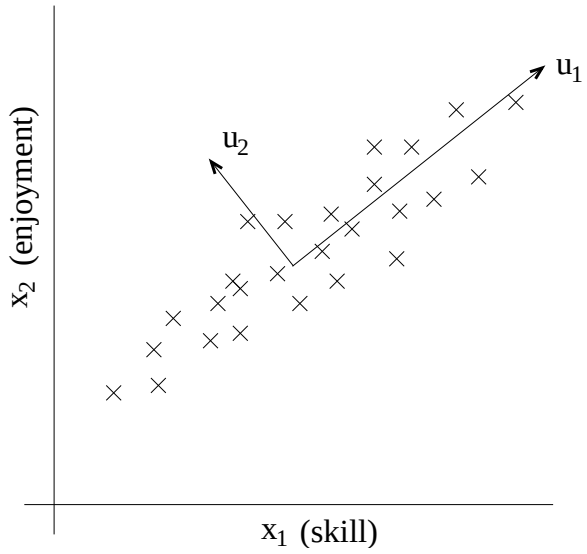
In our discussion of factor analysis, we gave a way to model data $x \in \mathbb{R}^n$ as "approximately" lying in some k-dimension subspace, where $k \ll n$. Specifically, we imagined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$, and then adding Ψ-covariance noise. Factor analysis is based on a probabilistic model, and parameter estimation used the iterative EM algorithm.

In this set of notes, we will develop a method, Principal Components Analysis (PCA), that also tries to identify the subspace in which the data approximately lies. However, PCA will do so more directly, and will require only an eigenvector calculation (easily done with the `eig` function in Matlab), and does not need to resort to EM.

Suppose we are given a dataset $\{x^{(i)}; i = 1, \ldots, m\}$ of attributes of m different types of automobiles, such as their maximum speed, turn radius, and so on. Let $x^{(i)} \in \mathbb{R}^n$ for each i ($n \ll m$). But unknown to us, two different attributes—some x_i and x_j—respectively give a car's maximum speed measured in miles per hour, and the maximum speed measured in kilometers per hour. These two attributes are therefore almost linearly dependent, up to only small differences introduced by rounding off to the nearest mph or kph. Thus, the data really lies approximately on an $n - 1$ dimensional subspace. How can we automatically detect, and perhaps remove, this redundancy?

For a less contrived example, consider a dataset resulting from a survey of pilots for radio-controlled helicopters, where $x_1^{(i)}$ is a measure of the piloting skill of pilot i, and $x_2^{(i)}$ captures how much he/she enjoys flying. Because RC helicopters are very difficult to fly, only the most committed students, ones that truly enjoy flying, become good pilots. So, the two attributes x_1 and x_2 are strongly correlated. Indeed, we might posit that that the

data actually likes along some diagonal axis (the u_1 direction) capturing the intrinsic piloting "karma" of a person, with only a small amount of noise lying off this axis. (See figure.) How can we automatically compute this u_1 direction?



We will shortly develop the PCA algorithm. But prior to running PCA per se, typically we first pre-process the data to normalize its mean and variance, as follows:

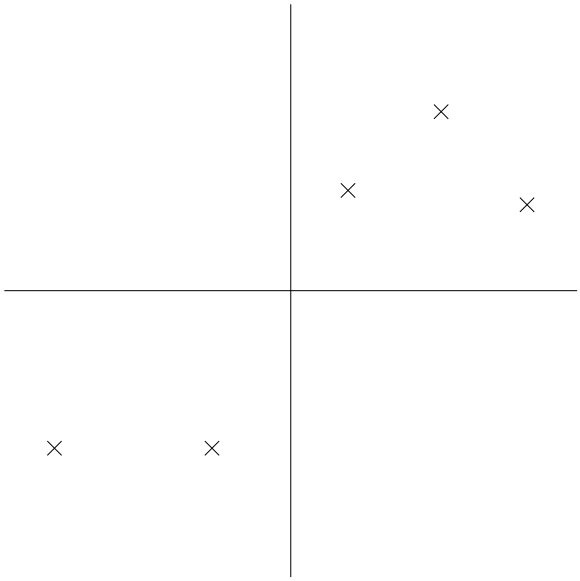
1. Let $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$.
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$.
3. Let $\sigma_j^2 = \frac{1}{m} \sum_i (x_j^{(i)})^2$
4. Replace each $x_j^{(i)}$ with $x_j^{(i)} / \sigma_j$.

Steps (1-2) zero out the mean of the data, and may be omitted for data known to have zero mean (for instance, time series corresponding to speech or other acoustic signals). Steps (3-4) rescale each coordinate to have unit variance, which ensures that different attributes are all treated on the same "scale." For instance, if x_1 was cars' maximum speed in mph (taking values in the high tens or low hundreds) and x_2 were the number of seats (taking values around 2-4), then this renormalization rescales the different attributes to make them more comparable. Steps (3-4) may be omitted if we had apriori knowledge that the different attributes are all on the same scale. One

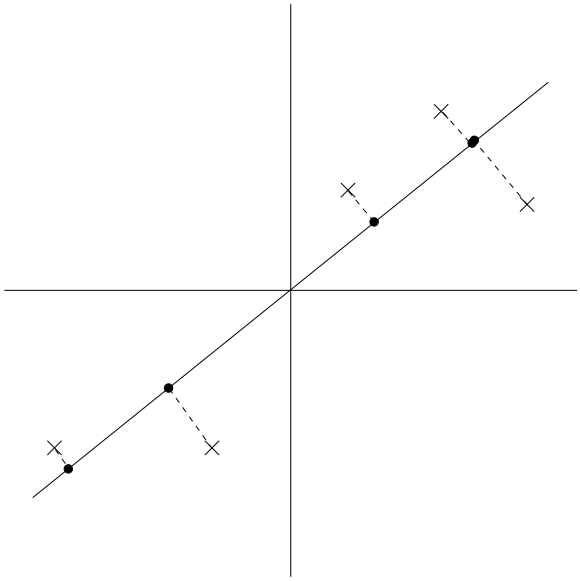
example of this is if each data point represented a grayscale image, and each $x_j^{(i)}$ took a value in $\{0, 1, \ldots, 255\}$ corresponding to the intensity value of pixel j in image i.

Now, having carried out the normalization, how do we compute the "major axis of variation" u—that is, the direction on which the data approximately lies? One way to pose this problem is as finding the unit vector u so that when the data is projected onto the direction corresponding to u, the variance of the projected data is maximized. Intuitively, the data starts off with some amount of variance/information in it. We would like to choose a direction u so that if we were to approximate the data as lying in the direction/subspace corresponding to u, as much as possible of this variance is still retained.

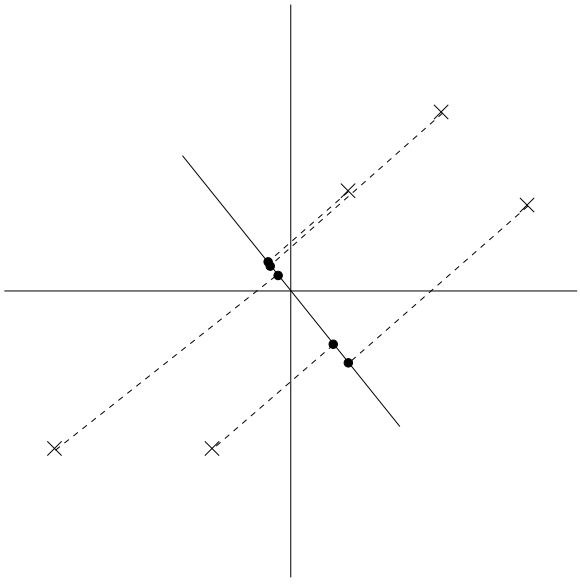
Consider the following dataset, on which we have already carried out the normalization steps:



Now, suppose we pick u to correspond the the direction shown in the figure below. The circles denote the projections of the original data onto this line.



We see that the projected data still has a fairly large variance, and the points tend to be far from zero. In contrast, suppose had instead picked the following direction:



Here, the projections have a significantly smaller variance, and are much closer to the origin.

We would like to automatically select the direction u corresponding to the first of the two figures shown above. To formalize this, note that given a

unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e., if $x^{(i)}$ is a point in our dataset (one of the crosses in the plot), then its projection onto u (the corresponding circle in the figure) is distance $x^T u$ from the origin. Hence, to maximize the variance of the projections, we would like to choose a unit-length u so as to maximize:

$$\begin{aligned}\frac{1}{m}\sum_{i=1}^{m}(x^{(i)^T}u)^2 &= \frac{1}{m}\sum_{i=1}^{m} u^T x^{(i)} x^{(i)^T} u \\ &= u^T\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T}\right)u.\end{aligned}$$

We easily recognize that the maximizing this subject to $||u||_2 = 1$ gives the principal eigenvector of $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T}$, which is just the empirical covariance matrix of the data (assuming it has zero mean).[1]

To summarize, we have found that if we wish to find a 1-dimensional subspace with with to approximate the data, we should choose u to be the principal eigenvector of Σ. More generally, if we wish to project our data into a k-dimensional subspace ($k < n$), we should choose $u_1, \ldots, u_k$ to be the top k eigenvectors of Σ. The u_i's now form a new, orthogonal basis for the data.[2]

Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ u_2^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k.$$

Thus, whereas $x^{(i)} \in \mathbb{R}^n$, the vector $y^{(i)}$ now gives a lower, k-dimensional, approximation/representation for $x^{(i)}$. PCA is therefore also referred to as a **dimensionality reduction** algorithm. The vectors $u_1, \ldots, u_k$ are called the first k **principal components** of the data.

Remark. Although we have shown it formally only for the case of $k = 1$, using well-known properties of eigenvectors it is straightforward to show that

[1] If you haven't seen this before, try using the method of Lagrange multipliers to maximize $u^T\Sigma u$ subject to that $u^T u = 1$. You should be able to show that $\Sigma u = \lambda u$, for some λ, which implies u is an eigenvector of Σ, with eigenvalue λ.

[2] Because Σ is symmetric, the u_i's will (or always can be chosen to be) orthogonal to each other.

of all possible orthogonal bases $u_1, \ldots, u_k$, the one that we have chosen maximizes $\sum_i ||y^{(i)}||_2^2$. Thus, our choice of a basis preserves as much variability as possible in the original data.

In problem set 4, you will see that PCA can also be derived by picking the basis that minimizes the approximation error arising from projecting the data onto the k-dimensional subspace spanned by them.

PCA has many applications; we will close our discussion with a few examples. First, compression—representing $x^{(i)}$'s with lower dimension $y^{(i)}$'s—is an obvious application. If we reduce high dimensional data to $k = 2$ or 3 dimensions, then we can also plot the $y^{(i)}$'s to visualize the data. For instance, if we were to reduce our automobiles data to 2 dimensions, then we can plot it (one point in our plot would correspond to one car type, say) to see what cars are similar to each other and what groups of cars may cluster together.

Another standard application is to preprocess a dataset to reduce its dimension before running a supervised learning learning algorithm with the $x^{(i)}$'s as inputs. Apart from computational benefits, reducing the data's dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting (e.g., linear classifiers over lower dimensional input spaces will have smaller VC dimension).

Lastly, as in our RC pilot example, we can also view PCA as a noise reduction algorithm. In our example it estimates the intrinsic "piloting karma" from the noisy measures of piloting skill and enjoyment. In class, we also saw the application of this idea to face images, resulting in **eigenfaces** method. Here, each point $x^{(i)} \in \mathbb{R}^{100\times100}$ was a 10000 dimensional vector, with each coordinate corresponding to a pixel intensity value in a 100x100 image of a face. Using PCA, we represent each image $x^{(i)}$ with a much lower-dimensional $y^{(i)}$. In doing so, we hope that the principal components we found retain the interesting, systematic variations between faces that capture what a person really looks like, but not the "noise" in the images introduced by minor lighting variations, slightly different imaging conditions, and so on. We then measure distances between faces i and j by working in the reduced dimension, and computing $||y^{(i)} - y^{(j)}||_2$. This resulted in a surprisingly good face-matching and retrieval algorithm.

Recurrent Neural Networks

Neural Computation : Lecture 12

1. Recurrent Neural Network Architectures
2. State Space Models and Dynamical Systems
3. Backpropagation Through Time
4. The NARX Model
5. Computational Power of Recurrent Networks
6. Applications – Reading Model, Associative Memory
7. Hopfield Networks and Boltzmann Machines

Recurrent Neural Network Architectures

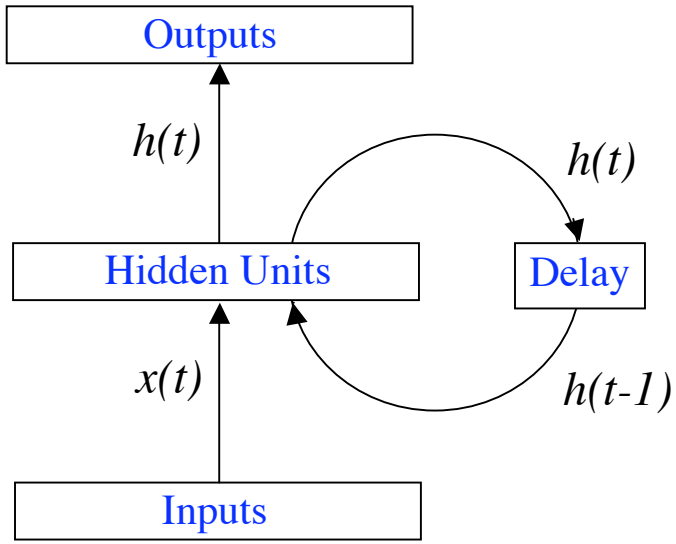
The fundamental feature of a ***Recurrent Neural Network (RNN)*** is that the network contains at least one ***feed-back connection***, so the activations can flow round in a loop. That enables the networks to do ***temporal processing*** and ***learn sequences***, e.g., perform sequence recognition/reproduction or temporal association/prediction.

Recurrent neural network architectures can have many different forms. One common type consists of a standard Multi-Layer Perceptron (MLP) plus added loops. These can exploit the powerful non-linear mapping capabilities of the MLP, and also have some form of ***memory***. Others have more uniform structures, potentially with every neuron connected to all the others, and may also have stochastic activation functions.

For simple architectures and deterministic activation functions, learning can be achieved using similar gradient descent procedures to those leading to the back-propagation algorithm for feed-forward networks. When the activations are stochastic, simulated annealing approaches may be more appropriate. The following will look at a few of the most important types and features of recurrent networks.

A Fully Recurrent Network

The simplest form of ***fully recurrent neural network*** is an MLP with the previous set of hidden unit activations feeding back into the network along with the inputs:



Note that the time *t* has to be ***discretized***, with the activations updated at each time step. The time scale might correspond to the operation of real neurons, or for artificial systems any time step size appropriate for the given problem can be used. A ***delay unit*** needs to be introduced to hold activations until they are processed at the next time step.

The State Space Model

If the neural network inputs and outputs are the vectors *x(t)* and *y(t)*, the three connection weight matrices are W_{IH}, W_{HH} and W_{HO}, and the hidden and output unit activation functions are f_H and f_O, the behaviour of the recurrent network can be described as a ***dynamical system*** by the pair of non-linear matrix equations:

$$h(t) = f_H\left(W_{IH}x(t) + W_{HH}h(t-1)\right)$$

$$y(t) = f_O\left(W_{HO}h(t)\right)$$

In general, the ***state*** of a dynamical system is a set of values that summarizes all the information about the past behaviour of the system that is necessary to provide a unique description of its future behaviour, apart from the effect of any external factors. In this case the state is defined by the set of hidden unit activations *h(t)*.

Thus, in addition to the input and output spaces, there is also a ***state space***. The ***order*** of the dynamical system is the dimensionality of the state space, the number of hidden units.

Stability, Controllability and Observability

Since one can think about recurrent networks in terms of their properties as dynamical systems, it is natural to ask about their *stability*, *controllability* and *observability*:

Stability concerns the boundedness over time of the network outputs, and the response of the network outputs to small changes (e.g., to the network inputs or weights).

Controllability is concerned with whether it is possible to control the dynamic behaviour. A recurrent neural network is said to be *controllable* if an initial state is steerable to any desired state within a finite number of time steps.

Observability is concerned with whether it is possible to observe the results of the control applied. A recurrent network is said to be *observable* if the state of the network can be determined from a finite set of input/output measurements.

A rigorous treatment of these issues is way beyond the scope of this module!

Learning and Universal Approximation

The ***Universal Approximation Theorem*** tells us that:

> Any non-linear dynamical system can be approximated to any accuracy by a recurrent neural network, with no restrictions on the compactness of the state space, provided that the network has enough sigmoidal hidden units.

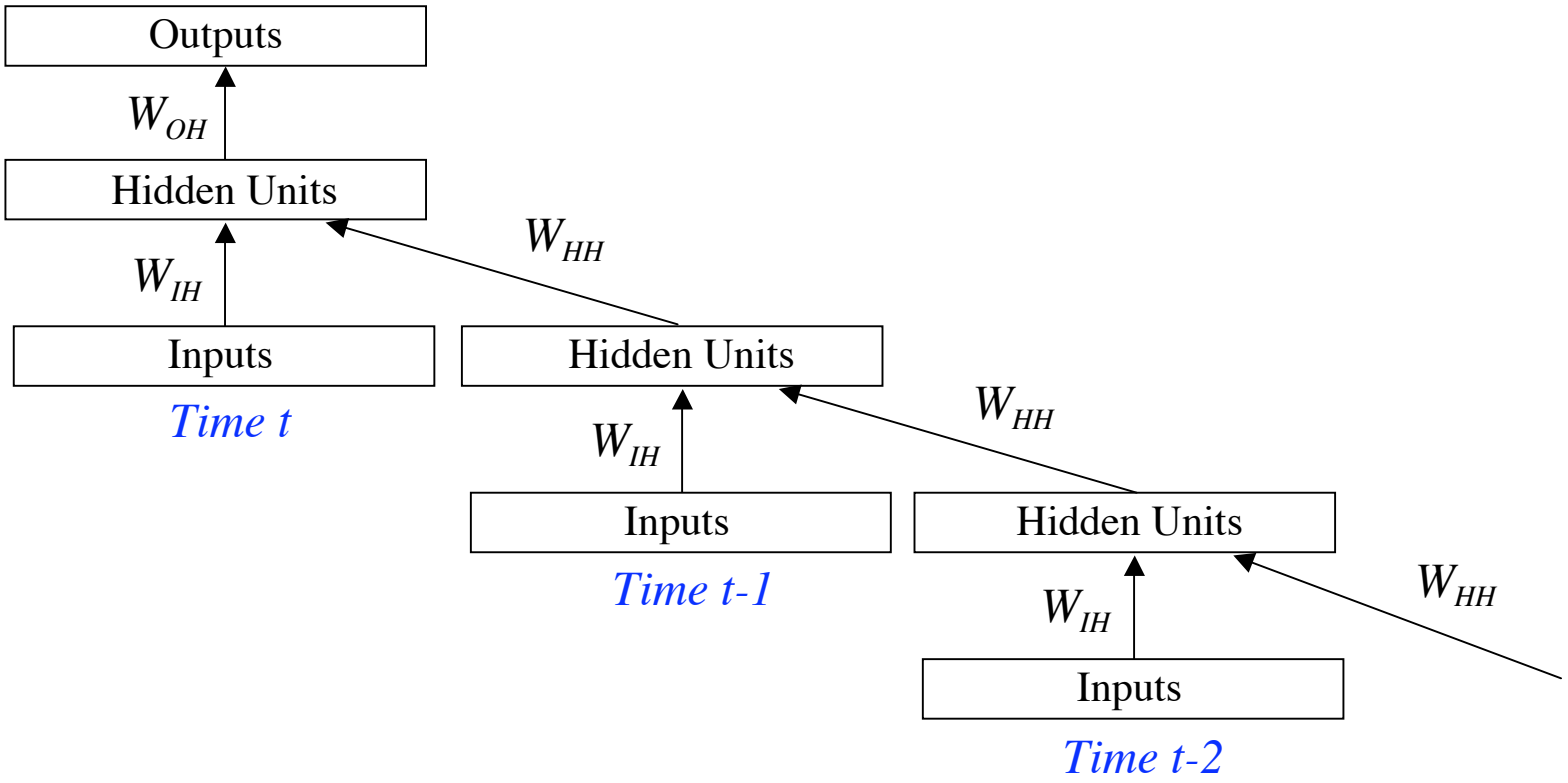
This underlies the computational power of recurrent neural networks.

However, knowing that a recurrent neural network can approximate any dynamical system does not tell us how to achieve it. As with feed-forward neural networks, we generally want them to learn from a set of training data to perform appropriately.

We can use ***Continuous Training***, for which the network state is never reset during training, or ***Epochwise Training***, which has the network state reset at each epoch. We now look at one particular learning approach that works for both training modes.

Unfolding Over Time

The recurrent network can be converted into a feed-forward network by ***unfolding over time***:



This means all the earlier theory about feed-forward network learning follows through.

Backpropagation Through Time

The Backpropagation Through Time ***(BPTT)*** learning algorithm is a natural extension of standard backpropagation that performs gradient descent on a complete unfolded network.

If a network training sequence starts at time t_0 and ends at time t_1, the total cost function is simply the sum over time of the standard error function $E_{sse/ce}(t)$ at each time-step:

$$E_{total}(t_0, t_1) = \sum_{t=t_0}^{t_1} E_{sse/ce}(t)$$

and the gradient descent weight updates have contributions from each time-step

$$\Delta w_{ij} = -\eta \frac{\partial E_{total}(t_0, t_1)}{\partial w_{ij}} = -\eta \sum_{t=t_0}^{t_1} \frac{\partial E_{sse/ce}(t)}{\partial w_{ij}}$$

The constituent partial derivatives $\partial E_{sse/ce} / \partial w_{ij}$ now have contributions from the multiple instances of each weight $w_{ij} \in \{W_{IH}, W_{HH}\}$ and depend on the inputs and hidden unit activations at previous time steps. The errors now have to be back-propagated through time as well as through the network.

Practical Considerations for BPTT

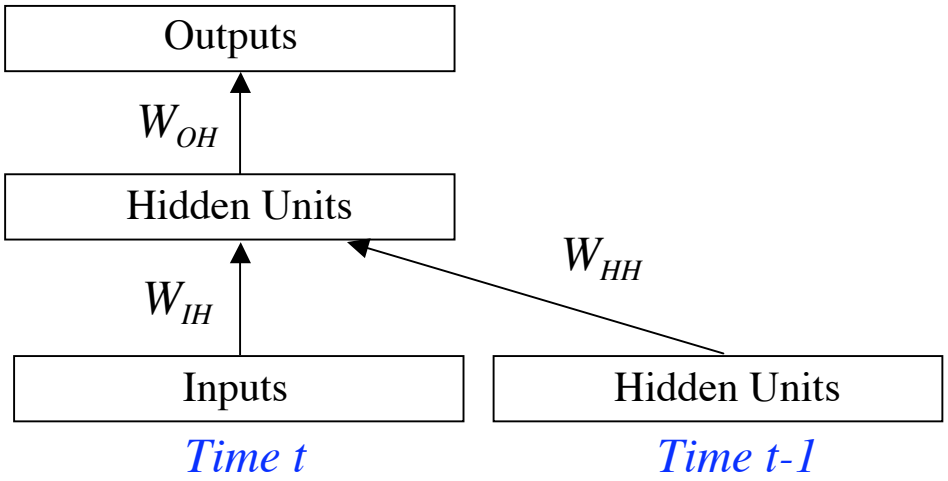
Even for the relatively simple recurrent network shown above, the unfolded network is quite complex, and keeping track of all the components at different points of time can become unwieldy. Most useful networks are even more problematic in this regard.

Typically the updates are made in an online fashion with the weights being updated at each time step. This requires storage of the history of the inputs and past network states at earlier times. For this to be computationally feasible, it requires ***truncation*** at a certain number of time steps, with the earlier information being ignored.

Assuming the network is stable, the contributions to the weight updates should become smaller the further back in time they come from. This is because they depend on higher powers of small feedback strengths (corresponding to the sigmoid derivatives multiplied by the feedback weights). This means that truncation is not as problematic as it sounds, though many times step may be needed in practice (e.g., ~30). Despite its complexity, BPTT has been shown many times to be an effective learning algorithm.

Simple Recurrent Networks

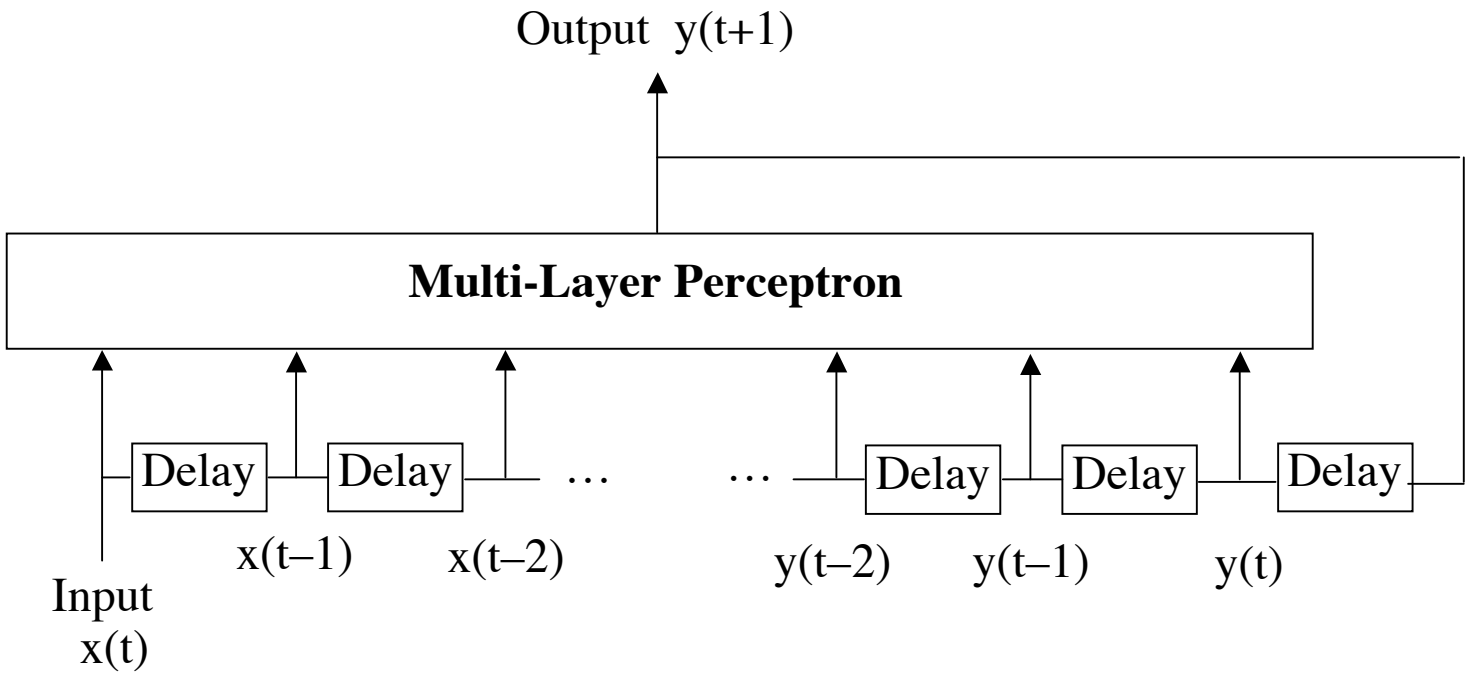
Truncating the unfolded network to just one time step reduces it to a so-called ***Simple Recurrent Network*** (which is also commonly known as an ***Elman network***):



In this case, each set of weights now only appears only once, so it is possible to apply the gradient descent approach using the standard backpropagation algorithm rather than full BPTT. This means that the error signal will not get propagated back very far, and it will be difficult for the network to learn how to use information from far back in time. In practice, this approximation proves to be too great for many practical applications.

The NARX Model

An alternative RNN formulation has a single input and a single output, with a delay line on the inputs, and the outputs fed back to the input by another delay line. This is known as the Non-linear Auto-Regressive with eXogeneous inputs (***NARX***) model:



It is particularly effective for ***Time Series Prediction***, where the target y(t+1) is x(t+1).

Computational Power of Recurrent Networks

Recurrent neural networks exemplified by the fully recurrent network and the NARX model have an inherent ability to simulate *finite state automata*. Automata represent abstractions of information processing devices such as computers. The computational power of a recurrent network is embodied in two main theorems:

Theorem 1

All Turing machines may be simulated by fully connected recurrent networks built of neurons with sigmoidal activation functions.

Proof: Siegelmann & Sontag, 1991, *Applied Mathematics Letters*, vol 4, pp 77-80.

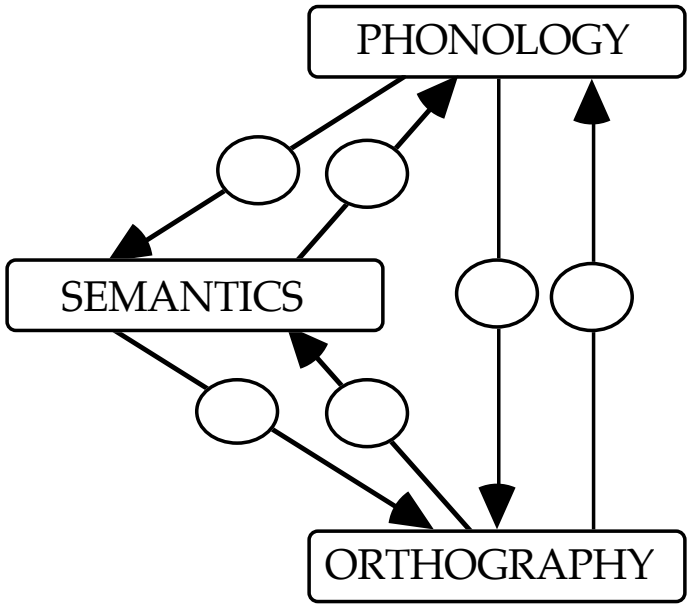
Theorem 2

NARX Networks with one layer of hidden neurons with bounded, one sided saturated (BOSS) activation functions and a linear output neuron can simulate fully connected recurrent networks with bounded one-sided saturated activation functions, except for a linear slowdown.

Proof: Siegelmann et al., 1997, *Systems, Man and Cybernetics, Part B*, vol 27, pp 208-215.

Application – Triangular Model of Reading

Any realistic ***model of reading*** must consider the interaction of orthography (letters), phonology (sounds) and semantics (meanings). A recurrent neural network of the form:



is required. The various sub-tasks (e.g., reading and spelling) need to be trained in line with human experience, and the dominant emergent pathways are found to depend on the nature of the language (e.g., English versus Chinese).

Application – Associative Memory

Associative Memory is the general idea of accessing memory through content rather than by address or location. One approach to this is for the memory content to be the pattern of activations on the nodes of a recurrent neural network. The idea is to start the network off with a pattern of activations that is a partial or noisy representation of the required memory content, and have the network settle down to the required content.

The basic associative memory problem can be stated as:

> Store a set of P binary valued patterns $\{\mathbf{t}^p\} = \{t_i^p\}$ in such a way that, when it is presented with a new pattern $\mathbf{s} = \{s_i\}$, the system (e.g., recurrent neural network) responds by producing whichever stored pattern $\mathbf{t}^p$ most closely resembles $\mathbf{s}$.

One could have separate input and output nodes to achieve this, or one could have a single set of nodes that act both as inputs and outputs. Either way, one needs to specify suitable node activation functions, and weight definition or update equations, so that the recurrent connections allow the learned memories to be accessed.

Hopfield Networks

The ***Hopfield Network*** ***or*** ***Hopfield Model*** is one good way to implement an associative memory. It is simply a fully connected recurrent network of N McCulloch-Pitts neurons. Activations are normally ±1, rather than 0 and 1, so the neuron activation equation is:

$$x_i = \text{sgn}\left(\sum_j w_{ij} x_j - \theta_i\right) \qquad \text{where} \qquad \text{sgn}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Unlike the earlier feed-forward McCulloch-Pitts networks, the activations here depend on time, because the activations keep changing till they have all settled down to some stable pattern. Those activations can be updated either ***synchronously*** or ***asynchronously***.

It can be shown that the required associative memory can be achieved by simply setting the weights w_{ij} and thresholds θ_j in relation to the target outputs $\mathbf{t}^p$ as follows:

$$w_{ij} = \frac{1}{N} \sum_{p=1}^{P} t_i^p t_j^p \qquad , \qquad \theta_i = 0$$

A stored pattern $\mathbf{t}^q$ will then be stable if the neuron activations are not changing, i.e.

$$t_i^q = \text{sgn}\left(\sum_j w_{ij} t_j^q - \theta_i\right) = \text{sgn}\left(\sum_j \frac{1}{N} \sum_p t_i^p t_j^p t_j^q\right)$$

which is best analyzed by separating out the q term from the p sum to give

$$t_i^q = \text{sgn}\left(t_i^q + \frac{1}{N} \sum_j \sum_{p \neq q} t_i^p t_j^p t_j^q\right)$$

If the second term in this is zero, it is clear that pattern number q is stable. It will also be stable if the second term is non-zero but has magnitude less than 1, because that will not be enough to move the argument over the step of the sign function sgn(x). In practice, this happens in most cases of interest as long as the number of stored patterns P is *small enough*. Moreover, not only will the stored patterns be stable, but they will also be ***attractors*** of patterns close by. Estimates of what constitutes a small enough number P leads to the idea of the ***storage capacity*** of a Hopfield network. A full discussion of Hopfield Networks can be found in most introductory books on neural networks.

Boltzmann Machines

A ***Boltzmann Machine*** is a variant of the Hopfield Network composed of N units with activations $\{x_i\}$. The state of each unit i is updated asynchronously according to the rule:

$$x_i = \begin{cases} +1 & \text{with probability } p_i \\ -1 & \text{with probability } 1 - p_i \end{cases}$$

where

$$p_i = \frac{1}{1 + \exp\left(-\left(\sum_{j=1}^{N} w_{ij} x_j - \theta_j\right) / T\right)}$$

with positive temperature constant T, network weights w_{ij} and thresholds θ_j.

The fundamental difference between the Boltzmann Machine and a standard Hopfield Network is the ***stochastic activation*** of the units. If T is very small, the dynamics of the Boltzmann Machine approximates the dynamics of the discrete Hopfield Network, but when T is large, the network visits the whole state space. Another difference is that the nodes of a Boltzmann Machine are split between visible input and output nodes, and hidden nodes, and the aim is to have the machine learn input-output mappings.

Training proceeds by updating the weights using the ***Boltzmann learning algorithm***

$$\Delta w_{ij} = -\frac{\eta}{T}\left(\langle x_i x_j \rangle_{fixed} - \langle x_i x_j \rangle_{free}\right)$$

where $\langle x_i x_j \rangle_{fixed}$ is the expected/average product of x_i and x_j during training with the input and output nodes fixed at a training pattern and the hidden nodes free to update, and $\langle x_i x_j \rangle_{free}$ is the corresponding quantity when the output nodes are also free.

For both Hopfield Networks and Boltzmann Machines one can define an ***energy function***

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} x_i x_j + \sum_{i=1}^{N} \theta_i x_i$$

and the network activation updates cause the network to settle into a local minimum of this energy. This implies that the stored patterns will be local minima of the energy. If a Boltzmann Machine starts off with a high temperature and is gradually cooled (known as ***simulated annealing***), it will tend to stay longer in the basins of attraction of the deepest minima, and have a good change of ending up in a global minimum at the end. Further details about Boltzmann Machines can be found in most introductory textbooks.

Restricted Boltzmann Machines and Deep Learning

One particularly important variation of the standard Boltzmann Machine is the ***Restricted Boltzmann Machine*** that has distinct sets of visible and hidden nodes, with no visible-to-visible or hidden-to-hidden connections allowed. This restriction makes it possible to formulate more efficient training algorithms, in particular, the ***Contrastive Divergence*** learning algorithm that performs an approximation to gradient descent.

These networks have proved effective in the field of ***Deep Learning***, and were what started much of the current interest in all forms of deep learning neural networks. Each layer of a many-layered feed-forward neural network can be efficiently pre-trained as an unsupervised Restricted Boltzmann Machine, and the whole network can then be fine-tuned using standard supervised Backpropagation.

Such use of semi-supervised feature learning in deep hierarchical neural networks is replacing hand-crafted feature creation in many areas. This is a major ongoing research area, with many new developments published each year.

Overview and Reading

1. We began by noting the important features of recurrent neural networks and their properties as fully fledged dynamical systems.
2. Then we studied a basic Fully Recurrent Network and unfolding it over time to give the Backpropagation Through Time learning algorithm.
3. This was followed by a brief overview of the NARX model and two theorems about the computational power of recurrent networks.
4. We looked at a Reading Model and Associative Memory as examples.
5. Finally, Hopfield Networks and Boltzmann Machines were introduced.

Reading

1. Haykin-2009: Sections 11.7, 13.7, 15.1 to 15.14
2. Hertz, Krogh & Palmer: Sections 2.1, 2.2, 7.1, 7.2, 7.3
3. Ham & Kostanic: Sections 5.1 to 5.9

CS229 Lecture notes

Andrew Ng

Part V
Support Vector Machines

This set of notes presents the Support Vector Machine (SVM) learning algorithm. SVMs are among the best (and many believe are indeed the best) "off-the-shelf" supervised learning algorithms. To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap." Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVMs efficiently in very high dimensional (such as infinite-dimensional) feature spaces, and finally, we'll close off the story with the SMO algorithm, which gives an efficient implementation of SVMs.

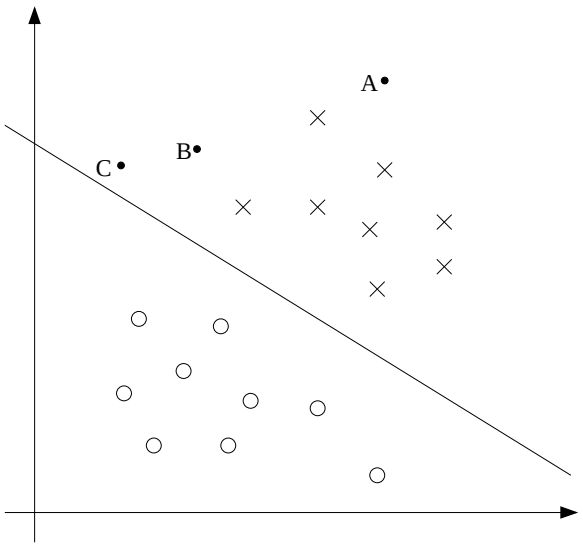
1 Margins: Intuition

We'll start our story on SVMs by talking about margins. This section will give the intuitions about margins and about the "confidence" of our predictions; these ideas will be made formal in Section 3.

Consider logistic regression, where the probability $p(y = 1|x; \theta)$ is modeled by $h_\theta(x) = g(\theta^T x)$. We would then predict "1" on an input x if and only if $h_\theta(x) \geq 0.5$, or equivalently, if and only if $\theta^T x \geq 0$. Consider a positive training example ($y = 1$). The larger $\theta^T x$ is, the larger also is $h_\theta(x) = p(y = 1|x; w, b)$, and thus also the higher our degree of "confidence" that the label is 1. Thus, informally we can think of our prediction as being a very confident one that $y = 1$ if $\theta^T x \gg 0$. Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\theta^T x \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^T x^{(i)} \gg 0$ whenever $y^{(i)} = 1$, and

$\theta^T x^{(i)} \ll 0$ whenever $y^{(i)} = 0$, since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which x's represent positive training examples, o's denote negative training examples, a decision boundary (this is the line given by the equation $\theta^T x = 0$, and is also called the **separating hyperplane**) is also shown, and three points have also been labeled A, B and C.



Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of y at A, it seems we should be quite confident that $y = 1$ there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it'd be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

2 Notation

To make our discussion of SVMs easier, we'll first need to introduce a new notation for talking about classification. We will be considering a linear classifier for a binary classification problem with labels y and features x. From now, we'll use $y \in \{-1, 1\}$ (instead of $\{0, 1\}$) to denote the class labels. Also, rather than parameterizing our linear classifier with the vector θ, we will use parameters w, b, and write our classifier as

$$h_{w,b}(x) = g(w^T x + b).$$

Here, $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise. This "w, b" notation allows us to explicitly treat the intercept term b separately from the other parameters. (We also drop the convention we had previously of letting $x_0 = 1$ be an extra coordinate in the input feature vector.) Thus, b takes the role of what was previously θ_0, and w takes the role of $[\theta_1 \ldots \theta_n]^T$.

Note also that, from our definition of g above, our classifier will directly predict either 1 or -1 (cf. the perceptron algorithm), without first going through the intermediate step of estimating the probability of y being 1 (which was what logistic regression did).

3 Functional and geometric margins

Let's formalize the notions of the functional and geometric margins. Given a training example $(x^{(i)}, y^{(i)})$, we define the **functional margin** of (w, b) with respect to the training example

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b).$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), we need $w^T x + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, we need $w^T x + b$ to be a large negative number. Moreover, if $y^{(i)}(w^T x + b) > 0$, then our prediction on this example is correct. (Check this yourself.) Hence, a large functional margin represents a confident and a correct prediction.

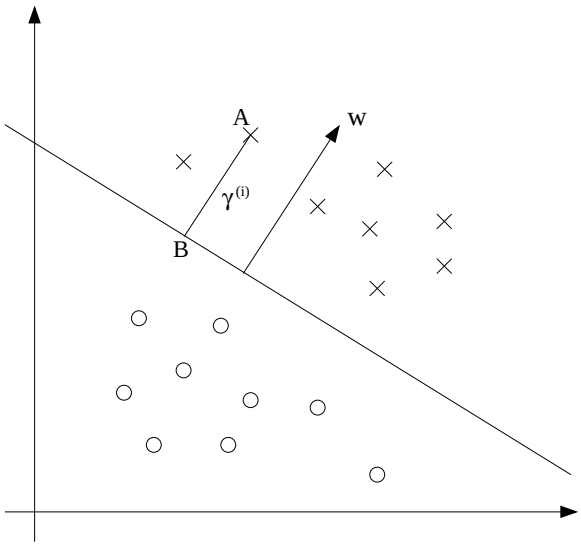
For a linear classifier with the choice of g given above (taking values in $\{-1, 1\}$), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace w with $2w$ and b with $2b$, then since $g(w^T x + b) = g(2w^T x + 2b)$,

this would not change $h_{w,b}(x)$ at all. I.e., g, and hence also $h_{w,b}(x)$, depends only on the sign, but not on the magnitude, of $w^T x + b$. However, replacing (w, b) with $(2w, 2b)$ also results in multiplying our functional margin by a factor of 2. Thus, it seems that by exploiting our freedom to scale w and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition such as that $||w||_2 = 1$; i.e., we might replace (w, b) with $(w/||w||_2, b/||w||_2)$, and instead consider the functional margin of $(w/||w||_2, b/||w||_2)$. We'll come back to this later.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the function margin of (w, b) with respect to S as the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}.$$

Next, let's talk about **geometric margins**. Consider the picture below:



The decision boundary corresponding to (w, b) is shown, along with the vector w. Note that w is orthogonal (at 90°) to the separating hyperplane. (You should convince yourself that this must be the case.) Consider the point at A, which represents the input $x^{(i)}$ of some training example with label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$, is given by the line segment AB.

How can we find the value of $\gamma^{(i)}$? Well, $w/||w||$ is a unit-length vector pointing in the same direction as w. Since A represents $x^{(i)}$, we therefore

find that the point B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. But this point lies on the decision boundary, and all points x on the decision boundary satisfy the equation $w^T x + b = 0$. Hence,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0.$$

Solving for $\gamma^{(i)}$ yields

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}.$$

This was worked out for the case of a positive training example at A in the figure, where being on the "positive" side of the decision boundary is good. More generally, we define the geometric margin of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right).$$

Note that if $||w|| = 1$, then the functional margin equals the geometric margin—this thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters; i.e., if we replace w with $2w$ and b with $2b$, then the geometric margin does not change. This will in fact come in handy later. Specifically, because of this invariance to the scaling of the parameters, when trying to fit w and b to training data, we can impose an arbitrary scaling constraint on w without changing anything important; for instance, we can demand that $||w|| = 1$, or $|w_1| = 5$, or $|w_1 + b| + |w_2| = 2$, and any of these can be satisfied simply by rescaling w and b.

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}.$$

4 The optimal margin classifier

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions

on the training set and a good "fit" to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a "gap" (geometric margin).

For now, we will assume that we are given a training set that is linearly separable; i.e., that it is possible to separate the positive and negative examples using some separating hyperplane. How will we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, \ldots, m \\ & ||w|| = 1. \end{aligned}$$

I.e., we want to maximize γ, subject to each training example having functional margin at least γ. The $||w|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ. Thus, solving this problem will result in (w, b) with the largest possible geometric margin with respect to the training set.

If we could solve the optimization problem above, we'd be done. But the "$||w|| = 1$" constraint is a nasty (non-convex) one, and this problem certainly isn't in any format that we can plug into standard optimization software to solve. So, let's try transforming the problem into a nicer one. Consider:

$$\begin{aligned} \max_{\hat{\gamma},w,b} \quad & \frac{\hat{\gamma}}{||w||} \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, \ldots, m \end{aligned}$$

Here, we're going to maximize $\hat{\gamma}/||w||$, subject to the functional margins all being at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \hat{\gamma}/||w|$, this will give us the answer we want. Moreover, we've gotten rid of the constraint $||w|| = 1$ that we didn't like. The downside is that we now have a nasty (again, non-convex) objective $\frac{\hat{\gamma}}{||w||}$ function; and, we still don't have any off-the-shelf software that can solve this form of an optimization problem.

Let's keep going. Recall our earlier discussion that we can add an arbitrary scaling constraint on w and b without changing anything. This is the key idea we'll use now. We will introduce the scaling constraint that the functional margin of w, b with respect to the training set must be 1:

$$\hat{\gamma} = 1.$$

Since multiplying w and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling w, b. Plugging this into our problem above, and noting that maximizing $\hat{\gamma}/||w|| = 1/||w||$ is the same thing as minimizing $||w||^2$, we now have the following optimization problem:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m \end{aligned}$$

We've now transformed the problem into a form that can be efficiently solved. The above is an optimization problem with a convex quadratic objective and only linear constraints. Its solution gives us the **optimal margin classifier**. This optimization problem can be solved using commercial quadratic programming (QP) code.[1]

While we could call the problem solved here, what we will instead do is make a digression to talk about Lagrange duality. This will lead us to our optimization problem's dual form, which will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

5 Lagrange duality

Let's temporarily put aside SVMs and maximum margin classifiers, and talk about solving constrained optimization problems.

Consider a problem of the following form:

$$\begin{aligned} \min_{w} \quad & f(w) \\ \text{s.t.} \quad & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

Some of you may recall how the method of Lagrange multipliers can be used to solve it. (Don't worry if you haven't seen it before.) In this method, we define the **Lagrangian** to be

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

[1] You may be familiar with linear programming, which solves optimization problems that have linear objectives and linear constraints. QP software is also widely available, which allows convex quadratic objectives and linear constraints.

Here, the β_i's are called the **Lagrange multipliers**. We would then find and set $\mathcal{L}$'s partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial w_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0,$$

and solve for w and β.

In this section, we will generalize this to constrained optimization problems in which we may have inequality as well as equality constraints. Due to time constraints, we won't really be able to do the theory of Lagrange duality justice in this class,[2] but we will give the main ideas and results, which we will then apply to our optimal margin classifier's optimization problem.

Consider the following, which we'll call the **primal** optimization problem:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{s.t.} \quad & g_i(w) \leq 0, \quad i = 1, \ldots, k \\ & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

To solve it, we start by defining the **generalized Lagrangian**

$$\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w).$$

Here, the α_i's and β_i's are the Lagrange multipliers. Consider the quantity

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \,:\, \alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{P}$" subscript stands for "primal." Let some w be given. If w violates any of the primal constraints (i.e., if either $g_i(w) > 0$ or $h_i(w) \neq 0$ for some i), then you should be able to verify that

$$\begin{aligned} \theta_{\mathcal{P}}(w) &= \max_{\alpha, \beta \,:\, \alpha_i \geq 0} f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w) \qquad (1) \\ &= \infty. \qquad (2) \end{aligned}$$

Conversely, if the constraints are indeed satisfied for a particular value of w, then $\theta_{\mathcal{P}}(w) = f(w)$. Hence,

$$\theta_{\mathcal{P}}(w) = \begin{cases} f(w) & \text{if } w \text{ satisfies primal constraints} \\ \infty & \text{otherwise.} \end{cases}$$

[2]Readers interested in learning more about this topic are encouraged to read, e.g., R. T. Rockarfeller (1970), Convex Analysis, Princeton University Press.

Thus, $\theta_{\mathcal{P}}$ takes the same value as the objective in our problem for all values of w that satisfies the primal constraints, and is positive infinity if the constraints are violated. Hence, if we consider the minimization problem

$$\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta),$$

we see that it is the same problem (i.e., and has the same solutions as) our original, primal problem. For later use, we also define the optimal value of the objective to be $p^* = \min_w \theta_{\mathcal{P}}(w)$; we call this the **value** of the primal problem.

Now, let's look at a slightly different problem. We define

$$\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{D}$" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α, β, here we are minimizing with respect to w.

We can now pose the **dual** optimization problem:

$$\max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta) = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta).$$

This is exactly the same as our primal problem shown above, except that the order of the "max" and the "min" are now exchanged. We also define the optimal value of the dual problem's objective to be $d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(w)$.

How are the primal and the dual problems related? It can easily be shown that

$$d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta) \leq \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta) = p^*.$$

(You should convince yourself of this; this follows from the "max min" of a function always being less than or equal to the "min max.") However, under certain conditions, we will have

$$d^* = p^*,$$

so that we can solve the dual problem in lieu of the primal problem. Let's see what these conditions are.

Suppose f and the g_i's are convex,[3] and the h_i's are affine.[4] Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i.

[3]When f has a Hessian, then it is convex if and only if the Hessian is positive semi-definite. For instance, $f(w) = w^T w$ is convex; similarly, all linear (and affine) functions are also convex. (A function f can also be convex without being differentiable, but we won't need those more general definitions of convexity here.)

[4]I.e., there exists a_i, b_i, so that $h_i(w) = a_i^T w + b_i$. "Affine" means the same thing as linear, except that we also allow the extra intercept term b_i.

Under our above assumptions, there must exist w^*, α^*, β^* so that w^* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and moreover $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$. Moreover, w^*, α^* and β^* satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\begin{aligned}
\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, n && (3)\\
\frac{\partial}{\partial \beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, l && (4)\\
\alpha_i^* g_i(w^*) &= 0, \quad i = 1, \ldots, k && (5)\\
g_i(w^*) &\leq 0, \quad i = 1, \ldots, k && (6)\\
\alpha^* &\geq 0, \quad i = 1, \ldots, k && (7)
\end{aligned}$$

Moreover, if some w^*, α^*, β^* satisfy the KKT conditions, then it is also a solution to the primal and dual problems.

We draw attention to Equation (5), which is called the KKT **dual complementarity** condition. Specifically, it implies that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. (I.e., the "$g_i(w) \leq 0$" constraint is **active**, meaning it holds with equality rather than with inequality.) Later on, this will be key for showing that the SVM has only a small number of "support vectors"; the KKT dual complementarity condition will also give us our convergence test when we talk about the SMO algorithm.

6 Optimal margin classifiers

Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

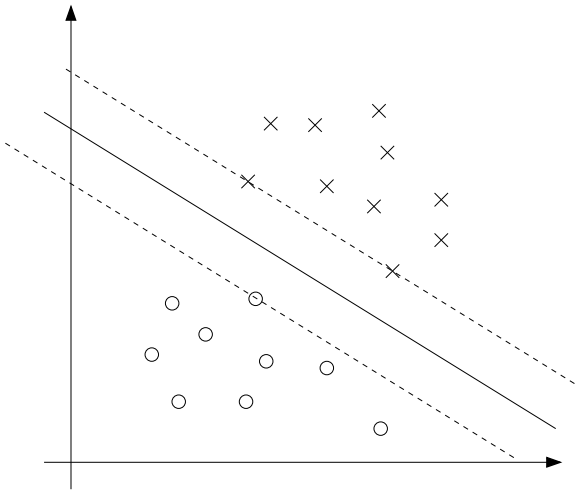
$$\begin{aligned}
\min_{w,b} \quad & \frac{1}{2}||w||^2 \\
\text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m
\end{aligned}$$

We can write the constraints as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones

corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



The points with the smallest margins are exactly the ones closest to the decision boundary; here, these are the three points (one negative and two positive examples) that lie on the dashed lines parallel to the decision boundary. Thus, only three of the α_i's—namely, the ones corresponding to these three training examples—will be non-zero at the optimal solution to our optimization problem. These three points are called the **support vectors** in this problem. The fact that the number of support vectors can be much smaller than the size the training set will be useful later.

Let's move on. Looking ahead, as we develop the dual form of the problem, one key idea to watch out for is that we'll try to write our algorithm in terms of only the inner product $\langle x^{(i)}, x^{(j)} \rangle$ (think of this as $(x^{(i)})^T x^{(j)}$) between points in the input feature space. The fact that we can express our algorithm in terms of these inner products will be key when we apply the kernel trick.

When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]. \tag{8}$$

Note that there're only "α_i" but no "β_i" Lagrange multipliers, since the problem has only inequality constraints.

Let's find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b (for fixed α), to get $\theta_{\mathcal{D}}$, which

we'll do by setting the derivatives of $\mathcal{L}$ with respect to w and b to zero. We have:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}. \tag{9}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{10}$$

If we take the definition of w in Equation (9) and plug that back into the Lagrangian (Equation 8), and simplify, we get

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{m} \alpha_i y^{(i)}.$$

But from Equation (10), the last term must be zero, so we obtain

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}.$$

Recall that we got to the equation above by minimizing $\mathcal{L}$ with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ (that we always had) and the constraint (10), we obtain the following dual optimization problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \end{aligned}$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations 3–7) to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's. We'll talk later

about the specific algorithm that we're going to use to solve the dual problem, but if we are indeed able to solve it (i.e., find the α's that maximize $W(\alpha)$ subject to the constraints), then we can use Equation (9) to go back and find the optimal w's as a function of the α's. Having found w^*, by considering the primal problem, it is also straightforward to find the optimal value for the intercept term b as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}. \tag{11}$$

(Check for yourself that this is correct.)

Before moving on, let's also take a more careful look at Equation (9), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^T x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (9), this quantity can also be written:

$$\begin{aligned} w^T x + b &= \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \right)^T x + b && (12) \\ &= \sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b. && (13) \end{aligned}$$

Hence, if we've found the α_i's, in order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate (13) and make our prediction.

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

7 Kernels

Back in our discussion of linear regression, we had a problem in which the input x was the living area of a house, and we considered performing regres-

sion using the features x, x^2 and x^3 (say) to obtain a cubic function. To distinguish between these two sets of variables, we'll call the "original" input value the input **attributes** of a problem (in this case, x, the living area). When that is mapped to some new set of quantities that are then passed to the learning algorithm, we'll call those new quantities the input **features**. (Unfortunately, different authors use different terms to describe these two things, but we'll try to use this terminology consistently in these notes.) We will also let ϕ denote the **feature mapping**, which maps from the attributes to the features. For instance, in our example, we had

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}.$$

Rather than applying SVMs using the original input attributes x, we may instead want to learn using some features $\phi(x)$. To do so, we simply need to go over our previous algorithm, and replace x everywhere in it with $\phi(x)$.

Since the algorithm can be written entirely in terms of the inner products $\langle x, z\rangle$, this means that we would replace all those inner products with $\langle \phi(x), \phi(z)\rangle$. Specifically, given a feature mapping ϕ, we define the corresponding **Kernel** to be

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

Then, everywhere we previously had $\langle x, z\rangle$ in our algorithm, we could simply replace it with $K(x, z)$, and our algorithm would now be learning using the features ϕ.

Now, given ϕ, we could easily compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. But what's more interesting is that often, $K(x, z)$ may be very inexpensive to calculate, even though $\phi(x)$ itself may be very expensive to calculate (perhaps because it is an extremely high dimensional vector). In such settings, by using in our algorithm an efficient way to calculate $K(x, z)$, we can get SVMs to learn in the high dimensional feature space space given by ϕ, but without ever having to explicitly find or represent vectors $\phi(x)$.

Let's see an example. Suppose $x, z \in \mathbb{R}^n$, and consider

$$K(x, z) = (x^T z)^2.$$

We can also write this as

$$\begin{aligned} K(x,z) &= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_j z_j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) \end{aligned}$$

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

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}.$$

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

For a related kernel, also consider

$$\begin{aligned} K(x,z) &= (x^T z + c)^2 \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) + \sum_{i=1}^{n} (\sqrt{2c}x_i)(\sqrt{2c}z_i) + c^2. \end{aligned}$$

(Check this yourself.) This corresponds to the feature mapping (again shown

for $n = 3$)

$$\phi(x) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \sqrt{2c}x_3 \\ c \end{bmatrix},$$

and the parameter c controls the relative weighting between the x_i (first order) and the x_ix_j (second order) terms.

More broadly, the kernel $K(x, z) = (x^Tz + c)^d$ corresponds to a feature mapping to an $\binom{n+d}{d}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$ that are up to order d. However, despite working in this $O(n^d)$-dimensional space, computing $K(x, z)$ still takes only $O(n)$ time, and hence we never need to explicitly represent feature vectors in this very high dimensional feature space.

Now, let's talk about a slightly different view of kernels. Intuitively, (and there are things wrong with this intuition, but nevermind), if $\phi(x)$ and $\phi(z)$ are close together, then we might expect $K(x, z) = \phi(x)^T\phi(z)$ to be large. Conversely, if $\phi(x)$ and $\phi(z)$ are far apart—say nearly orthogonal to each other—then $K(x, z) = \phi(x)^T\phi(z)$ will be small. So, we can think of $K(x, z)$ as some measurement of how similar are $\phi(x)$ and $\phi(z)$, or of how similar are x and z.

Given this intuition, suppose that for some learning problem that you're working on, you've come up with some function $K(x, z)$ that you think might be a reasonable measure of how similar x and z are. For instance, perhaps you chose

$$K(x, z) = \exp\left(-\frac{||x - z||^2}{2\sigma^2}\right).$$

This is a reasonable measure of x and z's similarity, and is close to 1 when x and z are close, and near 0 when x and z are far apart. Can we use this definition of K as the kernel in an SVM? In this particular example, the answer is yes. (This kernel is called the **Gaussian kernel**, and corresponds

to an infinite dimensional feature mapping ϕ.) But more broadly, given some function K, how can we tell if it's a valid kernel; i.e., can we tell if there is some feature mapping ϕ so that $K(x, z) = \phi(x)^T\phi(z)$ for all x, z?

Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ. Now, consider some finite set of m points (not necessarily the training set) $\{x^{(1)}, \ldots, x^{(m)}\}$, and let a square, m-by-m matrix K be defined so that its (i, j)-entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the **Kernel matrix**. Note that we've overloaded the notation and used K to denote both the kernel function $K(x, z)$ and the kernel matrix K, due to their obvious close relationship.

Now, if K is a valid Kernel, then $K_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T\phi(x^{(j)}) = \phi(x^{(j)})^T\phi(x^{(i)}) = K(x^{(j)}, x^{(i)}) = K_{ji}$, and hence K must be symmetric. Moreover, letting $\phi_k(x)$ denote the k-th coordinate of the vector $\phi(x)$, we find that for any vector z, we have

$$\begin{aligned}
z^T K z &= \sum_i \sum_j z_i K_{ij} z_j \\
&= \sum_i \sum_j z_i \phi(x^{(i)})^T \phi(x^{(j)}) z_j \\
&= \sum_i \sum_j z_i \sum_k \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
&= \sum_k \sum_i \sum_j z_i \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
&= \sum_k \left(\sum_i z_i \phi_k(x^{(i)}) \right)^2 \\
&\geq 0.
\end{aligned}$$

The second-to-last step above used the same trick as you saw in Problem set 1 Q1. Since z was arbitrary, this shows that K is positive semi-definite ($K \geq 0$).

Hence, we've shown that if K is a valid kernel (i.e., if it corresponds to some feature mapping ϕ), then the corresponding Kernel matrix $K \in \mathbb{R}^{m \times m}$ is symmetric positive semidefinite. More generally, this turns out to be not only a necessary, but also a sufficient, condition for K to be a valid kernel (also called a Mercer kernel). The following result is due to Mercer.[5]

[5]Many texts present Mercer's theorem in a slightly more complicated form involving L^2 functions, but when the input attributes take values in $\mathbb{R}^n$, the version given here is equivalent.

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

Given a function K, apart from trying to find a feature mapping ϕ that corresponds to it, this theorem therefore gives another way of testing if it is a valid kernel. You'll also have a chance to play with these ideas more in problem set 2.

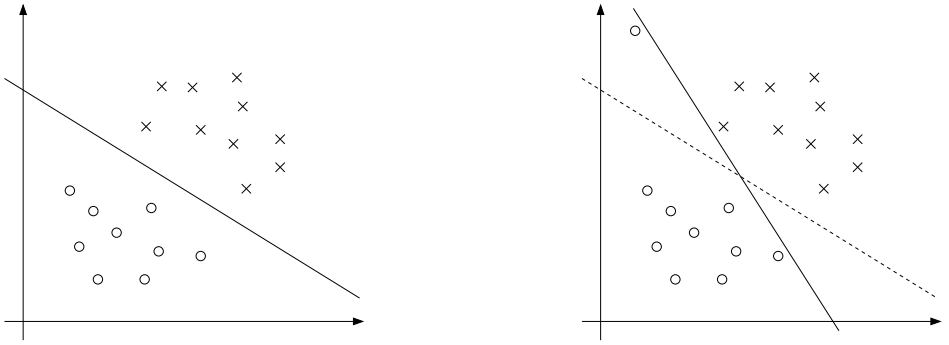
In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16x16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^T z)^d$ or the Gaussian kernel, SVMs were able to obtain extremely good performance on this problem. This was particularly surprising since the input attributes x were just 256-dimensional vectors of the image pixel intensity values, and the system had no prior knowledge about vision, or even about which pixels are adjacent to which other ones. Another example that we briefly talked about in lecture was that if the objects x that we are trying to classify are strings (say, x is a list of amino acids, which strung together form a protein), then it seems hard to construct a reasonable, "small" set of features for most learning algorithms, especially if different strings have different lengths. However, consider letting $\phi(x)$ be a feature vector that counts the number of occurrences of each length-k substring in x. If we're considering strings of English letters, then there are 26^k such strings. Hence, $\phi(x)$ is a 26^k dimensional vector; even for moderate values of k, this is probably too big for us to efficiently work with. (e.g., $26^4 \approx 460000$.) However, using (dynamic programming-ish) string matching algorithms, it is possible to efficiently compute $K(x, z) = \phi(x)^T \phi(z)$, so that we can now implicitly work in this 26^k-dimensional feature space, but without ever explicitly computing feature vectors in this space.

The application of kernels to support vector machines should already be clear and so we won't dwell too much longer on it here. Keep in mind however that the idea of kernels has significantly broader applicability than SVMs. Specifically, if you have any learning algorithm that you can write in terms of only inner products $\langle x, z \rangle$ between input attribute vectors, then by replacing this with $K(x, z)$ where K is a kernel, you can "magically" allow your algorithm to work efficiently in the high dimensional feature space corresponding to K. For instance, this kernel trick can be applied with the perceptron to derive a kernel perceptron algorithm. Many of the algorithms

that we'll see later in this class will also be amenable to this method, which has come to be known as the "kernel trick."

8 Regularization and the non-separable case

The derivation of the SVM as presented so far assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers. For instance, the left figure below shows an optimal margin classifier, and when a single outlier is added in the upper-left region (right figure), it causes the decision boundary to make a dramatic swing, and the resulting classifier has a much smaller margin.



To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization (using ℓ_1 **regularization**) as follows:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1,\ldots,m \\ & \xi_i \geq 0, \quad i = 1,\ldots,m. \end{aligned}$$

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

As before, we can form the Lagrangian:

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2} w^T w + C \sum_{i=1}^{m} \xi_i - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(x^T w + b) - 1 + \xi_i \right] - \sum_{i=1}^{m} r_i \xi_i.$$

Here, the α_i's and r_i's are our Lagrange multipliers (constrained to be ≥ 0). We won't go through the derivation of the dual again in detail, but after setting the derivatives with respect to w and b to zero as before, substituting them back in, and simplifying, we obtain the following dual form of the problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \end{aligned}$$

As before, we also have that w can be expressed in terms of the α_i's as given in Equation (9), so that after solving the dual problem, we can continue to use Equation (13) to make our predictions. Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (Equation 11 is no longer valid); see the comments in the next section/Platt's paper.

Also, the KKT dual-complementarity conditions (which in the next section will be useful for testing for the convergence of the SMO algorithm) are:

$$\alpha_i = 0 \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \tag{14}$$

$$\alpha_i = C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \leq 1 \tag{15}$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) = 1. \tag{16}$$

Now, all that remains is to give an algorithm for actually solving the dual problem, which we will do in the next section.

9 The SMO algorithm

The SMO (sequential minimal optimization) algorithm, due to John Platt, gives an efficient way of solving the dual problem arising from the derivation

of the SVM. Partly to motivate the SMO algorithm, and partly because it's interesting in its own right, let's first take another digression to talk about the coordinate ascent algorithm.

9.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

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

Here, we think of W as just some function of the parameters α_i's, and for now ignore any relationship between this problem and SVMs. We've already seen two optimization algorithms, gradient ascent and Newton's method. The new algorithm we're going to consider here is called **coordinate ascent**:

Loop until convergence: {

For $i = 1, \ldots, m$, {

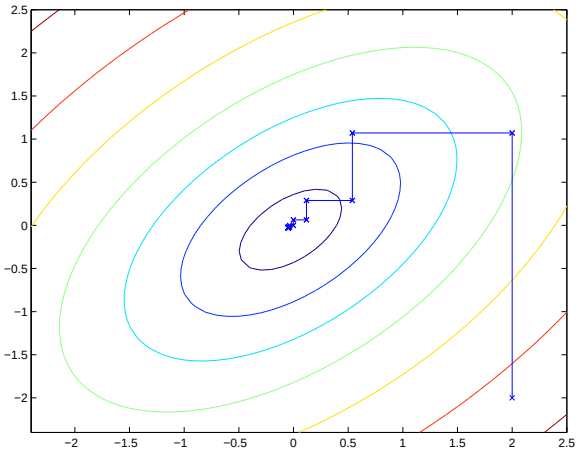
$\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_m)$.

}

}

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some α_i fixed, and reoptimize W with respect to just the parameter α_i. In the version of this method presented here, the inner-loop reoptimizes the variables in order $\alpha_1, \alpha_2, \ldots, \alpha_m, \alpha_1, \alpha_2, \ldots$. (A more sophisticated version might choose other orderings; for instance, we may choose the next variable to update according to which one we expect to allow us to make the largest increase in $W(\alpha)$.)

When the function W happens to be of such a form that the "arg max" in the inner loop can be performed efficiently, then coordinate ascent can be a fairly efficient algorithm. Here's a picture of coordinate ascent in action:



The ellipses in the figure are the contours of a quadratic function that we want to optimize. Coordinate ascent was initialized at $(2, -2)$, and also plotted in the figure is the path that it took on its way to the global maximum. Notice that on each step, coordinate ascent takes a step that's parallel to one of the axes, since only one variable is being optimized at a time.

9.2 SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm. Some details will be left to the homework, and for others you may refer to the paper excerpt handed out in class.

Here's the (dual) optimization problem that we want to solve:

$$\max_\alpha \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{17}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \tag{18}$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{19}$$

Let's say we have set of α_i's that satisfy the constraints (18-19). Now, suppose we want to hold $\alpha_2, \ldots, \alpha_m$ fixed, and take a coordinate ascent step and reoptimize the objective with respect to α_1. Can we make any progress? The answer is no, because the constraint (19) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^{m} \alpha_i y^{(i)}.$$

Or, by multiplying both sides by $y^{(1)}$, we equivalently have

$$\alpha_1 = -y^{(1)} \sum_{i=2}^{m} \alpha_i y^{(i)}.$$

(This step used the fact that $y^{(1)} \in \{-1, 1\}$, and hence $(y^{(1)})^2 = 1$.) Hence, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, \ldots, \alpha_m$ fixed, then we can't make any change to α_1 without violating the constraint (19) in the optimization problem.

Thus, if we want to update some subject of the α_i's, we must update at least two of them simultaneously in order to keep satisfying the constraints. This motivates the SMO algorithm, which simply does the following:

Repeat till convergence {

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

}

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 14-16) are satisfied to within some *tol*. Here, *tol* is the convergence tolerance parameter, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

The key reason that SMO is an efficient algorithm is that the update to α_i, α_j can be computed very efficiently. Let's now briefly sketch the main ideas for deriving the efficient update.

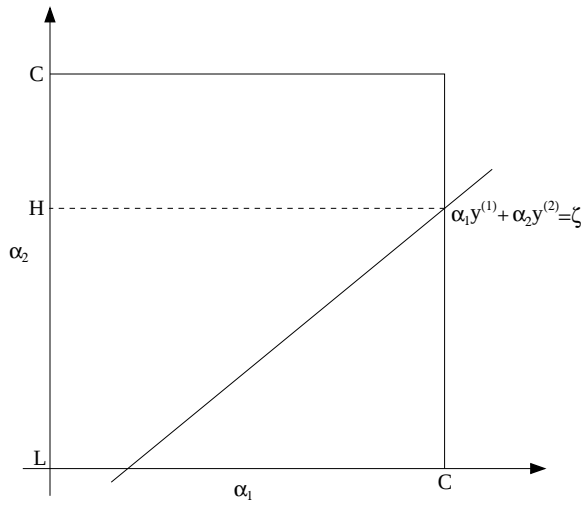
Let's say we currently have some setting of the α_i's that satisfy the constraints (18-19), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_m$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_m)$ with respect to α_1 and α_2 (subject to the constraints). From (19), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{m} \alpha_i y^{(i)}.$$

Since the right hand side is fixed (as we've fixed $\alpha_3, \ldots \alpha_m$), we can just let it be denoted by some constant ζ:

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \tag{20}$$

We can thus picture the constraints on α_1 and α_2 as follows:



From the constraints (18), we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Also plotted is the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$, on which we know α_1 and α_2 must lie. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$; otherwise, (α_1, α_2) can't simultaneously satisfy both the box and the straight line constraint. In this example, $L = 0$. But depending on what the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ looks like, this won't always necessarily be the case; but more generally, there will be some lower-bound L and some upper-bound H on the permissible values for α_2 that will ensure that α_1, α_2 lie within the box $[0, C] \times [0, C]$.

Using Equation (20), we can also write α_1 as a function of α_2:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}.$$

(Check this derivation yourself; we again used the fact that $y^{(1)} \in \{-1, 1\}$ so that $(y^{(1)})^2 = 1$.) Hence, the objective $W(\alpha)$ can be written

$$W(\alpha_1, \alpha_2, \ldots, \alpha_m) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, \ldots, \alpha_m).$$

Treating $\alpha_3, \ldots, \alpha_m$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints (18) (or, equivalently, that $L \leq \alpha_2 \leq H$), then we can easily maximize this quadratic function by setting its derivative to zero and solving. We'll let $\alpha_2^{new,unclipped}$ denote the resulting value of α_2. You should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{new,unclipped}$ and "clipping" it to lie in the

$[L, H]$ interval, to get

$$\alpha_2^{new} = \begin{cases} H & \text{if } \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & \text{if } L \leq \alpha_2^{new,unclipped} \leq H \\ L & \text{if } \alpha_2^{new,unclipped} < L \end{cases}$$

Finally, having found the α_2^{new}, we can use Equation (20) to go back and find the optimal value of α_1^{new}.

There're a couple more details that are quite easy but that we'll leave you to read about yourself in Platt's paper: One is the choice of the heuristics used to select the next α_i, α_j to update; the other is how to update b as the SMO algorithm is run.