Machine Learning

Comp3008

1 Introductory Lecture

a) Course Concepts

Learning “rules” from data
    More effective than building hard rules or expert systems
Newer approach to AI started in the 1980s
Scales much better and gives better results
Huge commercial applications

b) Aims

Concentrate on a few representative techniques
Theoretical framework for understanding different techniques
    maths: linear algebra, optimisation, probability, learning theory
Practical coursework

c) Learning from data

Tradition AI was rule based but newer techniques are based on learning from data
Supervised Learning (labelled data)
    Regression aka fitting functions
    Classification
Unsupervised (finding structures in data)
    Clustering (k-means)
    Data Reduction (PCA)
Reinforcement Learning (game playing)
Data is typically high dimensional meaning it has many attributes or features

2 Simple Classifiers

a) Classification

A standard task for machine learning
Typical scenario:

- some labelled data on which to train the data
- difficult to obtain the correct label
- we need a learning machine to perform the classification

Currently only concerned with binary classification

b) K-Nearest Neighbours

Simple but effective
Compute the distance from a new data point to each of the old ones:

$$d(x, x_k) = \sqrt{\sum_i (x_i - x_{ki})^2}$$

Find the K-Nearest Neighbours
Assign the class of the new data point to the majority class of the nearest neighbours
Increasing K means outliers become less significant but takes more time to compute
Increasing K also has a smoothing effect which may not be desirable but works well in high dimensions
KNN requires all features x_i to be the same scale
Can be helpful to normalise features of input and sample data
Typically normalise each 'axis' to be between 0 and 1
Can sometimes be useful to look at the variance on an axis (see slides)

c) Performance

Fast as there is no learning and it can be fairly accurate
Non-parametric algorithm

d) Perceptron

Prediction given by $f(x, w)$ where x is the input vector and w is 'weights' that we train
A classic parametric learning algorithm
Typically $f(x, w) = w^T x$ aka dotproduct of w and x
Also a response function (for the output) $g(x)$
Can have different response functions depending on the scenario:

- 'step' : for binary
- linear
- tanh

Perceptrons can be layered together so the output of one feeds as the input to another
Can use different types of inputs:

- local information in an image
- distance from 'centers' in the input space (RBF)
- Different polynormials (curve fitting)
- Layers of perceptrons (MLP)
- Eigenvectors of a kernel function (SVM)

One can also have multiple outputs the same set of input data

e) Linear Seperability

The perceptron with a step function performs classification
Can be visualised as a decision boundary in input space

- in 2D this can be thought of as a straight line on one side one class sits, on the other side a different class

The perceptron can only separate linear-separable inputs!
The perceptron is not powerful enough to do tasks such as parity (XOR) or connectedness
However, the world has a lot of linearly-seperable problems and these tend to be ones humans are good at
This is because data is in very high dimensions and the linear function needed gives a lot of flexibility
(see last years notes about perceptron learning)

3 Regression

a) Function Fitting

The concept is to fit a function to a set of data points as a learning machine
Want to minimize the errors
Unfortunately there an infinite set of functions that can go through the set of data points
So one has to reduce the functions being tested
However, simple functions are often better, complex ones often over learn the data

b) Least Squares Regression

$D=\{(x_k, y_k)\}$ Data points

$y_{predict}=f(x,w)$ where *w* is the parameters to learn and x is the input vector to get a value for

Want to minimise the error compared to example data

$\epsilon_k=f(x_k,w)-y_k$ this can be put into a vector

$E(x,w)=\|\epsilon\|^2$ *E(x, w)* is the objective function to minimise

c) Linear Least Squares Regression

Simplest functions to learn are linear ones
Aim is to define a plane in space whereby the distance for each training point to this plane is minimised

$f(x,w)=x^T w$

$\epsilon_k=x^T w-y_k$

can be combined into an error vector using a matrix X concatenating column data vectors together

$\epsilon=X^T w-y$

$E(x,w)=\|\epsilon\|^2$ this is the total error distance to be minimised

$E(x,w)=(X^T w-y)^T(X^T w-y)$

$E(x,w)=((X^T w)^T-y^T)(X^T w-y)$

$E(x,w)=(w^T X-y^T)(X^T w-y)$

$E(x,w)=w^T XX^T w-y^T X^T w-w^T Xy+y^T y$

$w^T Xy=y^T X^T w$ and is a number rather than matrix therefore

$E(x,w)=w^T XX^T w-2\mathrm{w}^T Xy+y^T y$

note this uses the trick of adding an extra feature
to cope with the bias
Minimum of this function (with respect to *w*) is when $\nabla E(x,w)=0$

$\nabla E(x,w)=\nabla\|\epsilon\|^2=\nabla\|X^T w-y\|^2=2(X^T w-y)$

this would mean $X^T w=y$ but if X is *MxN* and *M != N* then this can't be solved directly
Therefore must find a more explicit route!

$\nabla E(x,w)=\nabla(w^T XX^T w-2\mathrm{w}^T Xy+y^T y)$

$\nabla w^T XX^T w=(XX^T+(XX^T)^T)w$

$\nabla w^T XX^T w=(XX^T+XX^T)w$

$\nabla w^T XX^T w=2\mathrm{XX}^T w$

$\nabla 2\mathrm{w}^T Xy=2\mathrm{Xy}$

therefore

$\nabla E(x,w)=2(XX^T w-Xy)$

note due to $E(x,w)=\|\epsilon\|^2$ there is a single solution at Minima

$0=2(XX^T w-Xy)$

$0=XX^T w-Xy$

$Xy=XX^T w$

$(XX^T)^{-1}Xy=w$

This gives a direct solution for *w* in terms of the input and output values

d) Theory

For square matrix *X* this is a one-to-one mapping such that:

$y=X^T w$ and $w=(X^T)^{-1}y$

This is solvable and provides a good inverse mapping too

When we have more datapoints (examples) than parameters then generally we can not fit a curve through every point so *least squares* provides a solution to this by minimising the errors

this is an over constrained problem

If we have less datapoints than parameters the problem is under-constrained and therefore there will be many solutions of *w* for $y=X^T w$ and the problem is *illposed*

4 Generalisation

a) Why

Have learnt how to make learning machines which learn from example data (a training set)

However we want the machine to work for unseen data aswell

This is known as the *generalisation performance*

Usually the best learning is not the best generalisation

b) Generalisation Error

f(x) is the true function

p(x) is the probability of selecting *x*

Generalisation error defined as:

$E_g(w)=\int (f(x,w)-f(x))^2 p(x)\,dx$ i.e. the distance between the guess and the true function times by the probability of it occurring for every *x* summed

However, this is usually impossible to compute as we don't know *f(x)* otherwise we wouldn't be building a machine

It can be estimated though using some unseen data

unseen data $T=\{(x_k,y_k)\}_{k=1}^{T}$

$E_g(w)\approx E(w|T)=\frac{1}{T}\sum_{(x_k,y_k)\in T}(f(x_k,w)-y_k)^2$ *i.e. average error over the unseen set*

Important to *TEST* the generalisation performance using the validation set!

Can be used to help prevent over-fitting/over-learning of data which can easily occur in complex functions as one tries unnecessarily to go through every point with the learnt function

c) Ockham's Razor

Need to choose the simplest model that's not too simple

Principle referred to as Ockham's Razor

We should look for the simplest machine that can do the classification of data

We say that too complex machines over-fit the data
One way of formulating simplicity is *Bias-Variance*

d) Bias-Variance

Bias-Variance analysis shows that the generalisation error can be seen as the sum of two terms:

A bias due to the model being too simple

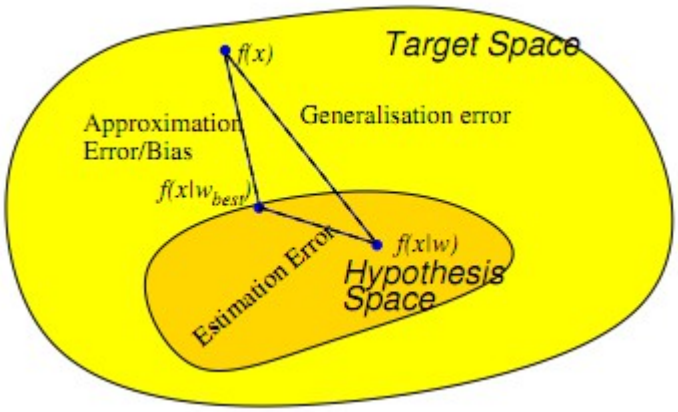
A variance due to the model sensitivity to the data (over fitted)

More of a theoretical process
Notation Aside: $\langle \ldots. \rangle_x$ means average over all x
Generalisation error is given by

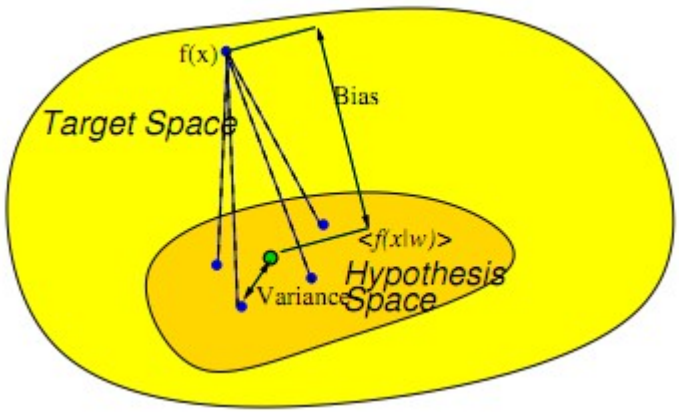
$$E_g(w) = \langle (f(x) - f(x,w))^2 \rangle_x$$

e) Approximation and Estimation Errors

f(x) is the target function in function space
f(x|w) is our guess at the function from training data *w*
f(x|w best) is the best possible guess at the function using the given structure for finding it (i.e. linear methods)



changing the data set used might bring us closer to the best estimate
Bias can be estimated by averaging the error of the machine after training on different data sets



One can find the average error for all data sets of the same size

$$\langle E_g(w) \rangle_D = \langle (f(x) - f(x,w))^2 \rangle_{D,x}$$

$$E_g = Bias + Variance$$

$$Bias = \langle (f(x) - \langle f(x,w) \rangle_D)^2 \rangle_x$$ i.e. the average error when we use the average learning machine

$Variance = \langle (f(x,w) - \langle f(x,w) \rangle_D)^2 \rangle_{D,x}$ measures the output variance as the machine is trained with different data:

current machine – average machine for all inputs and training sets

Bias : the average error against the average machine learnt
Variance: the average difference between the learnt machine and the average machine

Bias relates to how simple or complex a machine is and its ability to learn patterns/data
Variance relates to how likely the machine is to over fit the given data

f) Bias and Variance

Simple machines often have a high bias as they are too simple to learn the correct response
Complex machine often have a high variance as over fit their training data

g) How to reduce complexity

Choose simple machines
Reduce dimensionality of inputs (PCA)
Smoothing the input patterns
Pruning the network
Regularisation to boost simple functions

5 Data Sets and Pre-processing

a) Intro

Need to turn data into numerical vectors
Want useful examples and features which correlate otherwise it could be harmful for the learning machine
Preprocessing can be critical to getting machine learning to work

b) Categorical Data

Data is not necessarily numerical i.e. eye colour, education etc..
Have to turn them into numbers
Binary categories => 0/1 encoding
Multiple categories lead to two possibilities:

n features where only one is 1 : used when there is no ordering in values
n values for 1 feature : used when there is ordering in values

c) Normalising Data

Traditional vector length method can be used
Make each feature on a scale of 0 to 1

Normalise according to variance to the mean $x_{ki} \rightarrow \frac{x_{ki} - \mu_i}{\sigma_i}$

Required for Support Vector Machines to work well

d) Missing Data

Often the data sets will have some missing features
No one solution so it depends on the problem

- ignore missing examples
- replace missing features with mean of other datapoints
- flag examples and analyse later
- attempt to model the uncertainty probabilistically

e) Feature Selection

Incorporating useless features can be counter productive
Often achieve better performance by removing features
However this can be difficult to work out in advance
Exists weighting schemes for features based on correlation
For audio and image signals the number of features are too large and MUST be reduced
Can reduce complexity by (for example) :

searching for features in images like lines
PCA
…

f) Measuring Generalisation Performance

Want to measure the machine's performance on unseen data
Typically divide data in 3 sets:

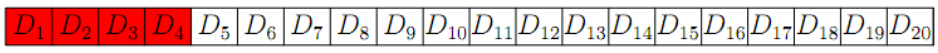
- training set
- parameter set (no always needed)
- test set

These must be independent !!
Estimating generalisation performance is important for choosing parameters
Can be hard to split up data if its in limited supply

g) Cross Validation

To estimate the generalisation error, perform the learning & testing multiple times on different parts of the data set

$$\mathcal{D} = \{D_i\}_{i=1}^{P} \quad D_i = (\mathbf{x}_i, y_i)$$



Test Set Training Set

5-fold cross-validation

$E_g =$ 5.1

$$\mathcal{D} = \{D_i\}_{i=1}^{P} \quad D_i = (\mathbf{x}_i, y_i)$$

D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}

Training Set Test Set Training Set

5-fold cross-validation

$$E_g = 3.7$$

$$\mathcal{D} = \{D_i\}_{i=1}^{P} \quad D_i = (\mathbf{x}_i, y_i)$$

D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}

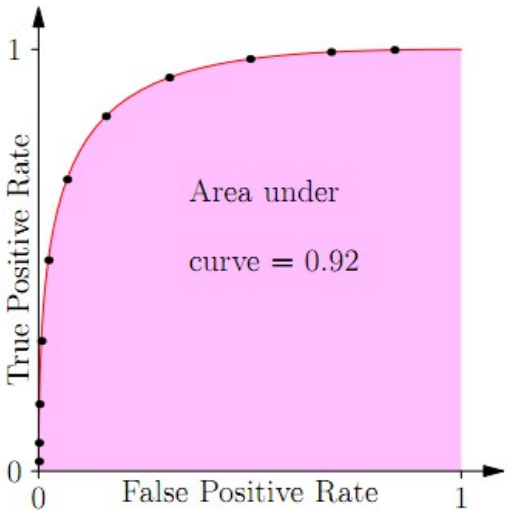
$$\langle E_g \rangle = \frac{5.1 + 3.7 + 4.6 + 4.6 + 3.3}{5} = 4.3$$

Can be expensive to run the cross validation in terms of computation but not in terms of time and gives fairly accurate results when there is little data

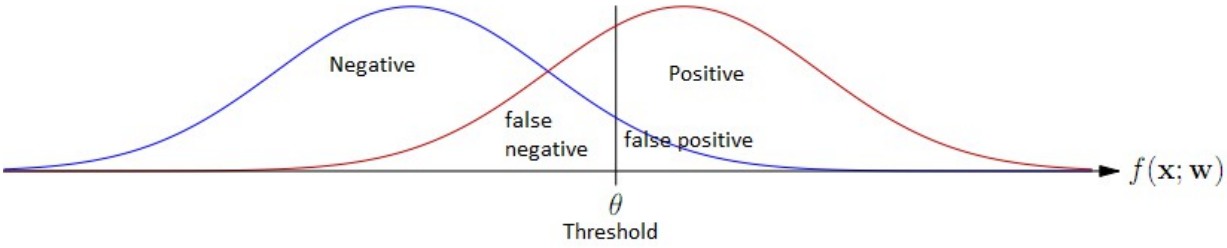
In K-fold cross-validation, the original sample is randomly partitioned into K subsamples. Of the K subsamples, a single subsample is retained as the validation data for testing the model, and the remaining K − 1 subsamples are used as training data. The cross-validation process is then repeated K times (the folds), with each of the K subsamples used exactly once as the validation data. The K results from the folds then can be averaged (or otherwise combined) to produce a single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once. 10-fold cross-validation is commonly used.

h) ROC Curves

Maps false positives again true positives



True Positives : where classifier is correct
False Positives : where classifier is wrong
A perfect classifier will have a flat (no gradient) curve



$$True\ Positive\ Rate = sensitivity = \frac{True\ Positive}{Positives} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$False\ Negative\ Rate = (1 - specifity) = \frac{False\ Positive}{Negative} = \frac{False\ Positive}{True\ Negative + False\ Positive}$$

6 Principle Component Analysis

a) What is it

Uses eigenvector maths to find the principle components (directions) of data in order to reduce feature size

Eigenvector for a matrix *M* are the vectors for which the matrix can be multiplied to which simple scale them. Matrices can either scale, translate or rotate vectors. A matrix will only scale its eigenvectors.

$Mv = \lambda v$ Where *M* is the matrix, *v* is an eigenvector and lambda is the eigenvalue which is the scaling value of the matrix

b) Feature Size

When we have many features it can be hard to use all the data

Learning becomes slow

Over-fitting becomes a big problem

would like to pre-process the data to help with this

PCA helps by removing data that doesn't provide much information

c) Spread

Often data varies in some limited directions

Reduces dimensions by projecting onto low dimensional subspace

d) Covariance Matrix

Data $D = \{x_k\}_{k=1}^{P}$

Mean $\mu = \frac{1}{P}\sum_{k=1}^{P} x_k$

Covariance matrix gives the variance in all different directions

Covariance matrix $C = \frac{1}{P-1}\sum_{k=1}^{P} (x_k - \mu)(x_k - \mu)^T$

Remember $x_k - \mu$ is a column vector, a column vector times a row vector forms a matrix therefore this can be simplified as:

$X = \frac{1}{\sqrt{P-1}}(x_1 - \mu, x_2 - \mu, \ldots, x_P - \mu)$ i.e. a Matrix formed of centred data points

$C = XX^T$

The covariance matrix is symmetric semi-definite which gives it certain properties

e) Properties of covariance matrix

Quadratic form of a vector and matrix is defined as $v^T Mv$

$v^T Cv = v^T XX^T v = u^T u = \|u\|^2 \geq 0$ where $u = X^T v$

Matrices with non-negative quadratic forms are known as positive semi-definite

Because its symmetric and square then for *n* features an *nxn* covariance matrix will be formed and *n* real orthogonal eigenvectors with real eigenvalues will be found

A covariance matrix will have a 0 eigenvalue only if there is no variation in the direction of the eigenvector

A covariance matrix will have 0 eigenvalues if there number of patterns is less than the number dimensions

A covariance matrix formed of *n+1* patterns with *n* dimensions will have no zero eigenvalues if the patterns are linearly independent

Matrices with no 0 eigenvalues are *full rank* and are also *positive definite* so we would expect that when *P>n* the covariance matrix will be positive definite

f) Orthogonal Matrices

Matrix *V* is orthogonal iff $V^{-1} = V^T$

This means that $V^{-1} V = V^T V = \text{Identity Matrix}$

g) Matrix Decomposition

Let *V* be a matrix formed of eigenvectors if a matrix *M*

$\Lambda = diag(\lambda_{1,..}\lambda_n)$

$M = M\ V\ V^T = V\ \Lambda\ V^T$

i.e. a Matrix *M* can be described in terms of its eigenvectors and eigenvalues

h) Eigenvalue Decomposition

The eigenvectors of *C* with the largest eigenvalues are known as the *principal components*

The eigenvalues are all positive real numbers

$Cv = \lambda v$

$v^T Cv = \lambda v^T v = \lambda \|v\|^2$

Since $v^T Cv \geq 0$ and $\|v\|^2 > 0$ then $\lambda \geq 0$

i) PCA Steps

1. Construct covariance matrix
2. Find eigenvectors and eigenvalues
3. Keep eigenvectors with largest eigenvalues (the principle components)
4. Project the inputs into the space spanned by the principle components

Then use projected inputs as inputs to our learning machine

j) Projection Matrix

To to step 4 one constructs a projection matrix

$P = \begin{pmatrix} v_1^T \\ v_2^T \\ v_m^T \end{pmatrix}$ where $m < n$ is the number of components we keep

Given an *n-dimensional* input pattern x we can construct an *m-dimensional* pattern z

$z = P(x - \mu)$

Then use z are our new pattern

k) Reconstruction

$\hat{x_k} = \mu + \sum_{i=1}^{m} z_i^k v_i$ Where $z_i^k = v_i^T (x_k - \mu)$ and eigenvectors are normalised

Reconstruction error $\|x_k - \hat{x_k}\|^2$ can be used to measure how much information has been lost

PCA finds a subspace with the smallest reconstruction error

7 PCA in practice

a) Images

When processing real images there are too many features to look at so it creates a very large covariance matrix

Mega pixel image → million wide matrix

P images span at most a *P-dimensional* subspace

This will be much smaller that the *NxN* space described by the images with standard PCA

The Dual Matrix 'trick' can be used to over come this

b) Dual Matrix

Typical covariance matrix is $C = XX^T$ consider the matrix $D = X^T X$

Suppose v is an eigen vector of D

$Dv = \lambda v$

$X^T X v = \lambda v$

$XX^T Xv = \lambda Xv$

$(XX^T) Xv = \lambda Xv$

$CXv = \lambda Xv$ let $u = Xv$

$Cu = \lambda u$

u is an eigen vector of C

c) Single Value Decomposition

Any *NxP* matrix X can be written as $X = USV^T$

Where:

U is an *NxN* orthogonal matrix

V is a *PxP* orthogonal matrix

S is an *NxP* diagonal matrix of singular values

A matrix M can also be described in terms of its eigenvectors and eigenvalues

$M = V \Lambda V^T$

d) Covariance matrix

$$C = XX^T$$
$$C = (USV^T)(USV^T)^T$$
$$C = (USV^T)(VS^T U^T)$$
$$C = USV^T VS^T U^T$$

Since V is orthogonal, $V^T V = I$

$$C = USS^T U^T$$

Since $M = V \Lambda V^T$ SS^T is the diagonal matrix of eigenvalues of C

$$D = X^T X$$
$$D = (USV^T)^T (USV^T)$$
$$D = (VS^T U^T)(USV^T)$$
$$D = VS^T U^T USV^T$$

Since U is orthogonal...

$$D = VS^T SV^T$$

So $S^T S$ becomes the diagonal matrix of eigenvalues of D

8 Matrix Math

A vector squared is its dotproduct with itself

$$v^2 = v \cdot v = v^T v$$

A transposed expression is the same as if each of its contents has been transposed

$$(X^T w - y)^T = (w^T X - y^T)$$
$$(BA)^T = A^T B^T$$

$$\nabla f(x) = \begin{pmatrix} \frac{\delta f(x)}{\delta x_1} \\ \dots \\ \frac{\delta f(x)}{\delta x_n} \end{pmatrix}$$

where $\frac{\delta f(x)}{\delta x_i}$ is the partial derivative meaning differentiate *f(x)* with respect to *xi* and treat all other features as constants

$$\nabla x^T M x = (M + M^T) x$$

if $M = M^T$ then $\nabla x^T M x = 2\text{Mx}$

9 Multi-Layer Perceptrons

a) Motivation

A normal perceptron can only solve linearly separable problems
Want to be able to build more complex learning machines
Still want to be able to train our machine
Perceptrons aren't very sophisticated

b) Non-Linear Perceptron

Learning machine with a single output $o = f(x|w)$ where typically

$f(x|w)=g(x^T w)$ and g is a squashing function for example $g(v)=\frac{1}{1+e^{-V+b}}$ where b is the bias

c) MLP

Combines layers of non-linear perceptrons together

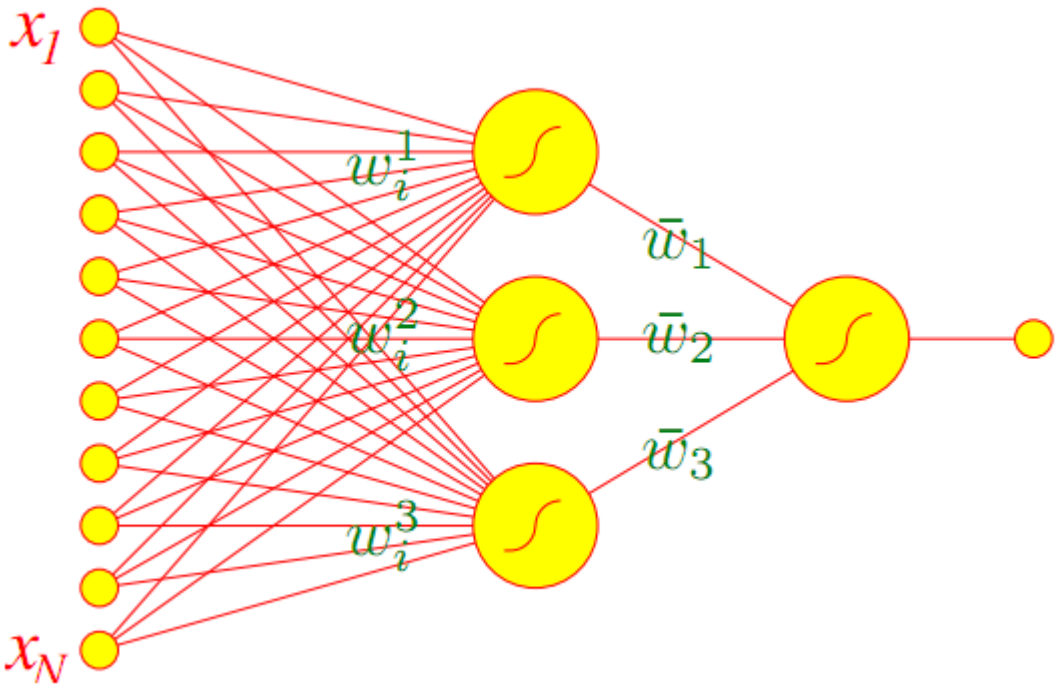


Illustration 1: A multilayered perceptron

$o=f(x|w)=g(\bar{x}^T \bar{w})$

$\bar{x}^T \bar{w}=\sum_i \bar{w}_i \bar{x}_i$

$\bar{x}_i=g(x^T w_i)$ where *wi* is the weight vector for the *ith* perceptron in the 1st layer

This gives a final answer of

$o=g(\sum_i \bar{w}_i g(x^T w_i))$ and of course can be extended for more layers as necessary

d) Multi-layer Linear Perceptrons

Has the same power as a normal perceptron with more complexity so don't use!

e) Non-linear least squares

Given some data $D=\{(x_k, y_k)\}_{k=1}^{P}$

We can define the error as:

$E(w|D)=\frac{1}{|D|}\sum_{(x_k,y_k)\in D}(y_k-f(x_k|w))^2$

This may be a highly non-linear function of the weights

Reduce error rate (learning) by optimisation, one technique of which is hill climbing

$E(w|D)=\frac{1}{P}\sum_{(x_k,y_k)\in D}(y_k-f(x_k|w))^2$

$$\nabla E(w|D)=\frac{2}{P}\sum_{(x_k,y_k)\in D}(y_k-f(x_k|w))\nabla f(x_k|w)$$

Let $f(x_k|w)-y_k=\delta_k$

Using chain rule for function *u* which takes a single value and *v* which takes a vector:

$$\nabla_w u(v(\boldsymbol{w}))=u'(v(\boldsymbol{w}))\nabla_w v(\boldsymbol{w})$$

therefore: $\nabla_w g(x^T w)=g'(x^T w)\nabla_w(x^T w)=g'(x^T w)x$

$$\nabla E(w|D)=\frac{2}{P}\sum_{(x_k,y_k)\in D}\delta_k g'(x_k^T w)x_k$$

for a logistic perceptron:

$$g(v)=\frac{1}{1+e^{-v}} \quad \text{so} \quad g'(v)=\frac{e^{-v}}{(1+e^{-v})^2}$$

for a tanh perceptron:

$$g(v)=\tanh(v) \quad \text{so} \quad g'(v)=1-\tanh^2(v)$$

For back-propagation one iteratively learns by minimising the error rate:

$w_{new}=w-\eta\nabla E(w|(x_k,y_k))$ where η is the learning rate (there are other and better ways of optimisation than this)

f) Online and Batch learning

Online learning : learn 1 example at a time, minimising its error rate
Batch learning : train on all examples at the same time (as mentioned above)

Online learning:

$$E(w|(x_y,y_k))=(f(x_k|w)-y_k)^2$$
$$\nabla_w E(w|(x_y,y_k))=2(f(x_k|w)-y_k)\nabla_w f(x_k|w)$$

10 Radial Basis Functions

a) What are they

They are functions which depend on the distance between one point and another
if $\Phi(x)=\Phi(\|x\|)$ then its a radial function

b) Examples

Gaussian

$$\Phi(x)=e^{-\beta x}$$

Multi quadric

$$\Phi(x)=\sqrt{x^2+\beta^2}$$

http://en.wikipedia.org/wiki/Radial_basis_function for more examples

c) Combining

Different functions can be combined in order to estimate another function
Each radial-function will be given a centre μ_i and a 'radius' σ_i
Regression can be estimated like so:

$$y_k=f(x_k|w)=BIAS+\sum_{i=1}^{N}w_i\Phi\left(\frac{\|x-\mu_i\|}{\sigma_i}\right)$$

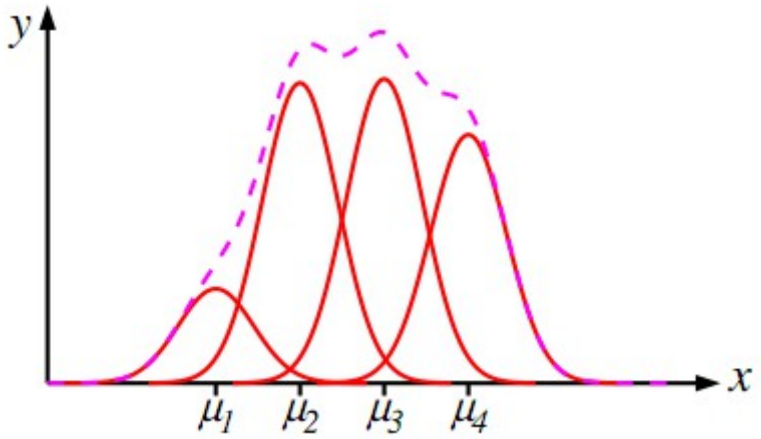


Illustration 2: Example output of a radial basis function. Note the height of the functions is the weight.

This can be combined further to produce a network of nodes and functions with multiple outputs.

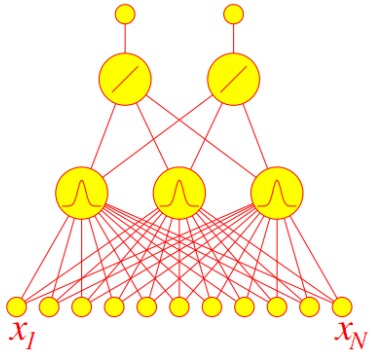


Illustration 3: Radial Basis network. Note the top 2 nodes are simply linear perceptrons.

d) Finding centers

Radial basis functions depend on the distance to centres
Finding these centres and how many are needed can be hard
They are normally found in clusters

e) k-Means Clustering

Find *k* clusters in the data
Can be tricky to find the right *k*
Easy algorithm and popular

Takes a set of unlabelled data $D=\{x_j\}_1^N$

Algorithm:

1. Choose K
2. Randomly assign each data point to a cluster C_i

3. Do until no change:

   Calculate mean of the cluster $\mu_i = \frac{1}{|C_i|} \sum_{k \in C_i} x_k$

   Assign each data point to its nearest mean

The means of the final clusters can then be used as centres in RBF's

f) Choosing widths

If widths are too small they'll be lots of peaks
If widths are too big then they'll be no sharp transitions
One way to find the widths is to calculate the average distance

$$\sigma_i^2 = \frac{1}{P-1} \sum_{j!=i} \|\mu_j - \mu_i\|^2$$

11 Optmisation

a) Gradient Optmisation

Maxima or minima of f occurs when $\nabla f(x) = 0$
The gradient points towards local maxima and minima

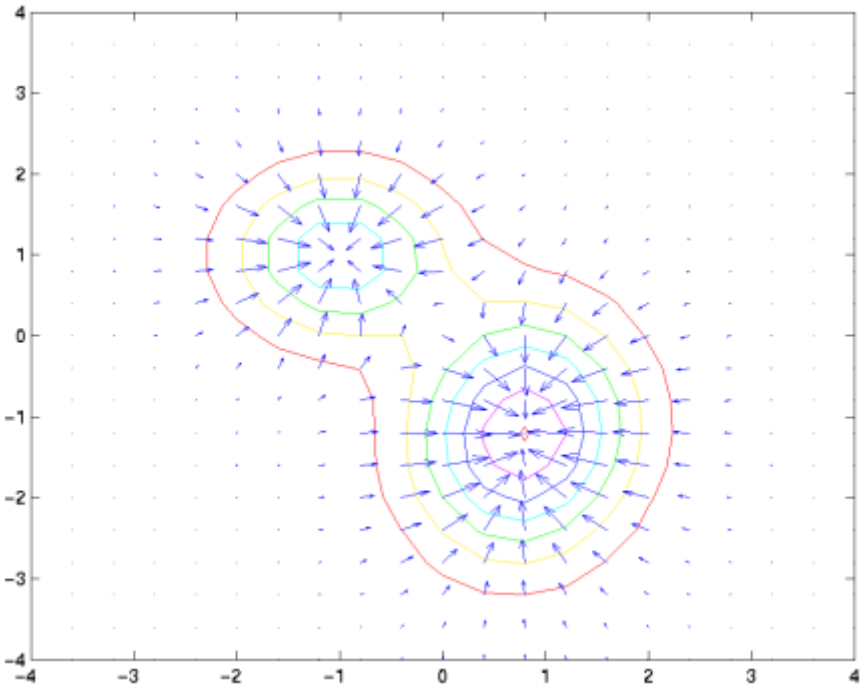
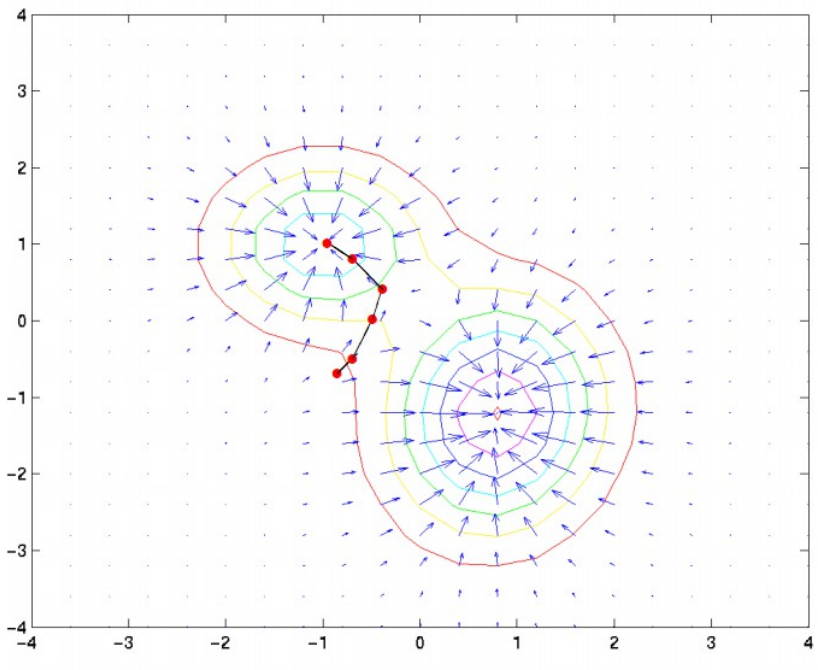


Illustration 4: Contours pointing towards minimas and maximas

For linear and other simple functions this can be found explicitly but for others it has to be done iteratively by 'hill climbing' or 'gradient descent'

hill climbing : finding maximas $x_{new} = x - \eta \nabla(x)$
gradient descent : finding minimas $x_{new} = x - \eta \nabla f(x)$



b) Newton's Method

Morse's theorem says that nearly all minima's are quadratic
Taylor Series:

$$f(x+h)=f(x)+f'(x)h+\frac{f''(x)h^2}{2}...$$

let x be the minima and h be our current guess
this says that if we're close enough to x (i.e. x-h is small) then higher terms can be ignored
Differentiating with respect to h gives the h which minimises this function

$$0=f'(x)+f''(x)h$$

$$h=\frac{-f'(x)}{f''(x)}$$

which suggests the iteration scheme

$$x=x_n-\frac{f'(x)}{f''(x)}$$

Converges quadratically if we start close enough to the minima
In higher dimensions:

$x=x_n-H^{-1}\nabla f(x)$ where H^{-1} is the hessian matrix of the function

c) More gradient descent

$x=x_0-\eta\nabla f(x_0)$ Need to choose a good learning rate/step size η
Too small and it takes a lot of time
Too large and it can over shoot the minima

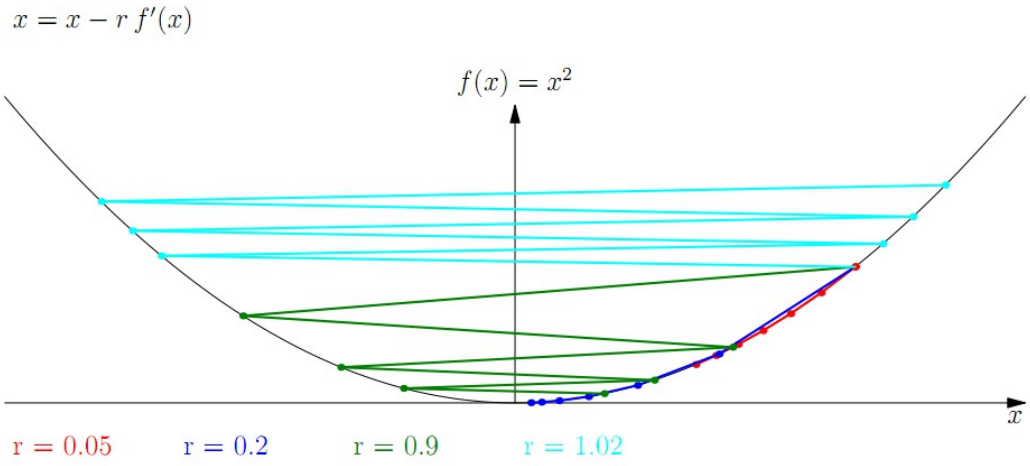


Illustration 5: When step size is too big, overshooting

One way is to find the η which minimises $f(x_0 - \eta \nabla f(x_0))$

d) Other algorithms

Conjuate gradient
Levenberg-Marquardt

12 Regularisation

a) What is

One way to improve generalisation performance is to bias a learning machine to learn simpler/smoother functions
This should help reduce the sensitivity of the learning machine on the learning data
To achieve this one can add *regularisation* terms that punish complex functions

b) Example

Using linear-least squares regression
let Φ be a matrix of input vectors, *w* be the vector of weights, *y* be a column vector of expected outputs

$E = \|\Phi^T w - y\|^2 + vR(w)$

R(w) is the regularisation function and *v* is its weight

$E = \|\Phi^T w - y\|^2 + v\|w\|^2$ this is known as *weight decay* as it punishes functions with lots of weights. *E* is the error which we want to minimise

$E = \|\Phi^T w - y\|^2 + v\|w\|^2$

$E = w^T \Phi\Phi^T w - 2w^T \Phi y + y^T y + vw^T w$

$E = w^T (\Phi\Phi^T + vI) w - 2w^T \Phi y + y^T y$ where *I* is the identity matrix

$\nabla E = 2(\Phi\Phi^T + vI) w - 2\Phi y$

minimum when $\nabla E = 0$

so best *w* is : $\hat{w} = 2(\Phi\Phi^T + vI)^{-1}\Phi y$

Previously the best weight was $\hat{w} = 2(\Phi\Phi^T)^{-1}\Phi y$ so now with regularisation the minima depends less on the data. This means although the result won't fit the test data so well it will be less sensitive to the data

c) Tikhonov Regularisation

More direct penalty for non-smooth functions
Looks at the second derivative of input patterns

$$\frac{1}{P}\sum_{p=1}^{P}\sum_{i=1}^{N}\left(\frac{\delta^2 f(x_p;w)}{\delta x_i^2}\right)^2$$ (average second derivative over all input patterns)

d) Summary

In machine learning we care about unseen data performance
Over complicated machines don't give very good generalisation performance
Wish to tailor the complexity of the machine to the data set
One way to do this automatically is by introducing regularisation terms

13 Computational Learning Theory

a) Classification

Can use different rules to categorise data
Most rules may be spurious (useless and false)

> i.e. in separating red & green triangles & squares into 2 categories then the machine could learn to separate on shape when its meant to separate on colour

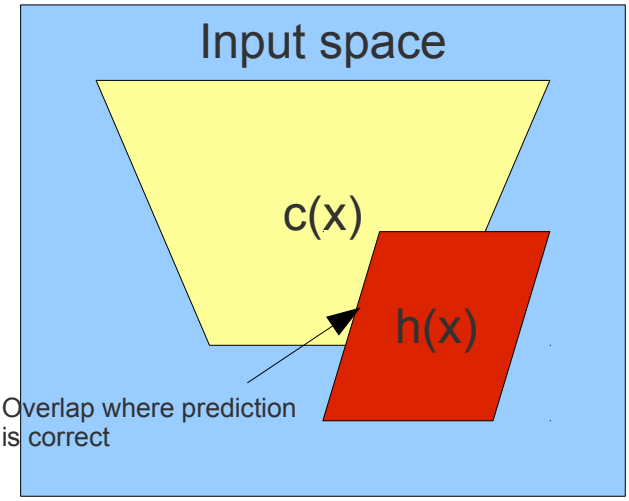
Increasing the data set size will reduce the amount of spurious rules
But how much data is needed?

b) PAC Learning

Probably Approximately Correct learning
Tries to answer the data-set size question
Only assumption is that the example data has the same probability distribution as the real data
More complex machines can find more complex rules so these need to be trimmed away
Probably : only as good as the examples shown
Approximately : want to give good predictions but not necessarily perfect

c) Generalisation Error

The generalisation error for a hypothesis *h(x)* is
$E(h|p)=P_p(h(x)!=c(x))$ where *p* is the probability distribution of x's and *c(x)* is the true answer
Essentially its the probability of making an error



d) PAC-Learnable

A concept class C is PAC learn-able if for all concepts in it $c \in C$ and a probability distribution p a learner choosing a hypothesis $h \in H$ can:

achieve a generalisation performance of $E(h|p) < \epsilon$ for $0 < \epsilon < \frac{1}{2}$

with a probability of $1-\delta$ and $0 < \delta < \frac{1}{2}$

using P training examples

where P is a polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$

The idea is to restrict the number of hypothesis's that can be learnt by eliminating those with a generalisation error greater than ϵ . $1-\delta$ refers to the certainty that the machine hasn't learnt a hypothesis with an error greater than ϵ .

e) Finite Consistent Hypothesis Spaces

Suppose the hypothesis space is finite and contains the true answer

Suppose our learning machine is able to correctly categorise all the training examples

$$E(h|p) = 0$$

Consider examples $D = \{(x_k, c(x_k))\}_{k=1}^{P}$

Consider a hypothesis with an errror $E(h|p) > \epsilon$ this means that the probability of it getting all the training examples correct is: $(1-\epsilon)^P$

There will be $k < |H|$ hypothesises with $E(h|p) > \epsilon$

The probability of any of these being correct means marginalising over all the k hypothesis

$$k(1-\epsilon)^P < |H|(1-\epsilon)^P \leq |H|e^{-\epsilon P}$$

Using $\delta \geq |H|e^{-\epsilon P}$ the number of training examples needed to get a generalisation error better than ϵ with a probability of $1-\delta$ is given by:

$$\delta \geq |H|e^{-\epsilon P}$$
$$\ln(\delta) - \ln(|H|) \geq -\epsilon P$$
$$\ln(|H|) - \ln(\delta) \leq \epsilon P$$
$$\frac{1}{\epsilon}\ln(|H|) - \ln(\delta) \leq P$$

$P > \frac{1}{\epsilon}(\ln(|H|) + \ln(\frac{1}{\delta}))$

Example:

for a hypothesis space of 2^{30} and an error of 1% is required with a probability of $1-10^{-6}$ then $P = \frac{1}{0.01}(\ln(2^{30}) + \ln(\frac{1}{10^{-6}})) = 3461$

f) Size of the hypothesis space

Suppose our features are *n* binary attributes and hypothesis's are a conjunction of the features
i.e. $h(x) = x_1 \wedge x_2 \wedge \neg x_3 \wedge .... \wedge x_n$
This gives a total of 2^n possible hypothesis
A more sensible hypothesis might combine different conjunctions
i.e. $h(x) = (x_1 \wedge \neg x_4 \wedge x_5) \vee (x_2 \wedge \neg x_1 \wedge \neg x_3) ...$
if there were *k* clauses each with 3 variables then $|H| = (2n)^{3k}$

g) Unlearnable hypothesis spaces

More complex learning machines let you learn more complex rules but also require more training examples to guarantee success
If a learning machine was complex enough to be able to identify every input then there would be $|H| = 2^{|X|}$ where its binary classification and *X* is the input set
$P > \frac{1}{\epsilon}(\ln(|H|) + \ln(\frac{1}{\delta}))$ and depends mostly on *H*
this means that $P \approx \ln(|H|) \approx |X|$ so the machine would have to see every example
This theory reinforces the idea that small learning machines are better

h) Inconsistent learners

Previous analysis assumed a correct hypothesis could be found
This is rarely true especially if the hypothesis space is small
So instead we ask 'how many examples are needed to get a generalisation error within ϵ with a certainty of $1-\delta$ '

$P > \frac{1}{2\epsilon^2}(\ln(|H|) + \ln(\frac{1}{\delta}))$

Unfortunately this now means that *P* grows with $\frac{1}{\epsilon^2}$

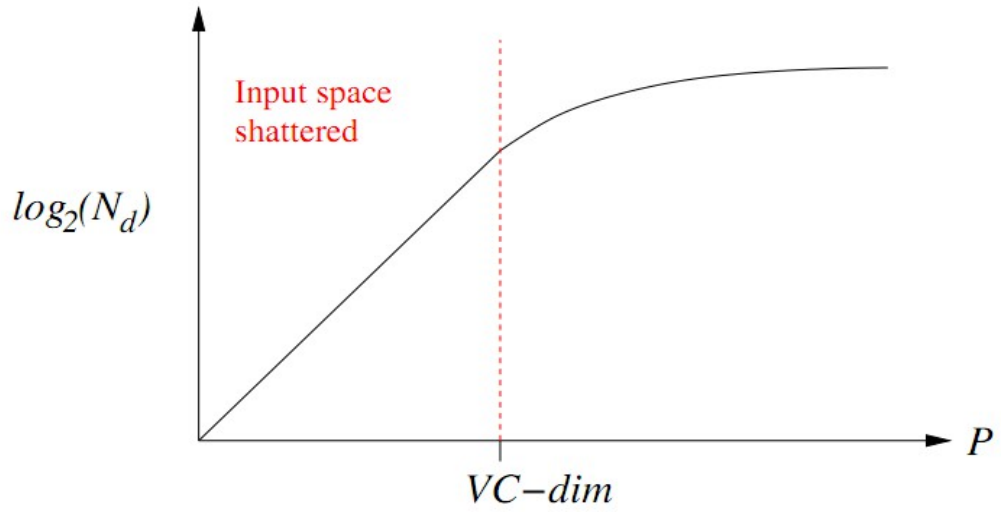
i) Infinite Hypothesis Spaces

Machines using continuous weights have infinite hypothesis space
Theory can be extended for this case though

j) Dichotomy

Consider a machine that performs binary classification
A dichotomy is one particular way of splitting the inputs into 2 classes
for *P* input patterns there are 2^P dichotomies

Let N_d be the number of Dichotomies a particular machine can produce



The VC-Dim defines a number of inputs
Above the VC-Dim the generalisation performance starts to improve
Below this the input space is shattered i.e. every input can be classified
this gives poor generalisation
VC-Dim : Vapnik-Chervonenkis Dimension
The result for the number of samples is similar for that for finite spaces

$$P \geq \frac{1}{\epsilon}(4\log_2(\frac{3}{\delta}) + 8\mathrm{VC}(H)\log_2(\frac{13}{\epsilon}))$$

The VC(-Dim) depends on the machine in question
for a (linear) perceptron it is $N+1$ where N is the number of features

k) Strenghts and Weakness of PAC

Provides worst case bounds on performance
Typical performance is normally much better than the worst case so one can get away with less training samples
Regularisation can make finding the terms needed harder
PAC has provided a major step forward in machine learning and contributed to SVMs

14 Optimisation

a) Constrained Optmisation

Suppose we have the problem $\min_x f(x)$ subject to $g(x)=0$
Standard procedure is to express this as a lagrangian

$$L(x,\alpha) = f(x) - \alpha g(x)$$

Then in the extended space find

$$\max_{\alpha} \min_{x} L(x,\alpha)$$

$$\nabla_x L = \nabla_x f(x) - \alpha \nabla_x g(x)$$

$$\nabla_\alpha L = g(x)$$

both of these terms are 0 at the solution so

$$\nabla_x f(x) = \alpha \nabla_x g(x) \quad \text{and}$$

$$g(x) = 0$$

Example:
minimise $f(x)=x^2+2y^2+xy$ subject to $g(x)=x-2y-3=0$

$\nabla_x f(x)=\alpha \nabla_x g(x)$ so

$$\begin{pmatrix} 2x-y \\ -x+4y \end{pmatrix}=\alpha\begin{pmatrix} 1 \\ -2 \end{pmatrix}$$

and

$g(x)=x-2y-3=0$

Then solve the simultaneous equations

b) Multiple Constraints

Suppose we have the problem $\min_x f(x) \text{ subject to } g_k(x)=0 \text{ for } k=1\ldots N$

$$L(x,\alpha)=f(x)-\sum_{k=1}^{N}\alpha_k g_k(x)$$

this means that

$$\nabla_x f(x)=\sum_{k=1}^{N}\alpha_k \nabla_x g_k(x)$$

Using the other conditions, one solves the simultaneous equations for the system

c) Inequality constraints

$\min_x f(x) \text{ subject to } g(x)\geq 0$

Two things can happen:

- minimum of *f(x)* satisfies $g(x)>0$ giving an unconstrained optimisation problem (finding a point in a region)
- OR minimum of *f(x)* satisfies $g(x)=0$ giving a constrained problem (finding a point a long some boundary)

Create $L(x,\alpha)=f(x)-\alpha g(x)$ and minimise as normal

if $\alpha=0$ then solution is in the region else if $\alpha>0$ then it lies on a boundary created by the function $g(x)=0$

Known as KKT conditions

15 Support Vector Machines

a) What are they

Relatively new learning machine type

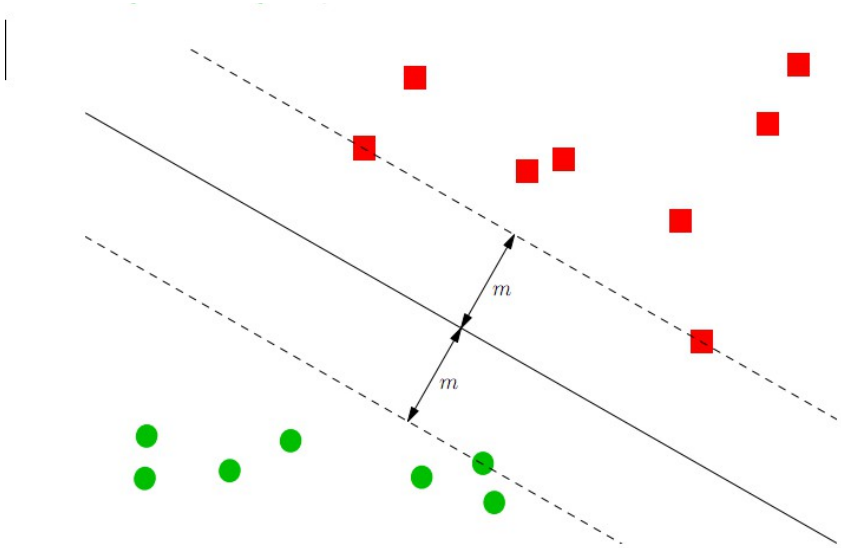
Get very competitive results on many problems

Many different forms (classification, regression etc..) but simplest type performs binary classification

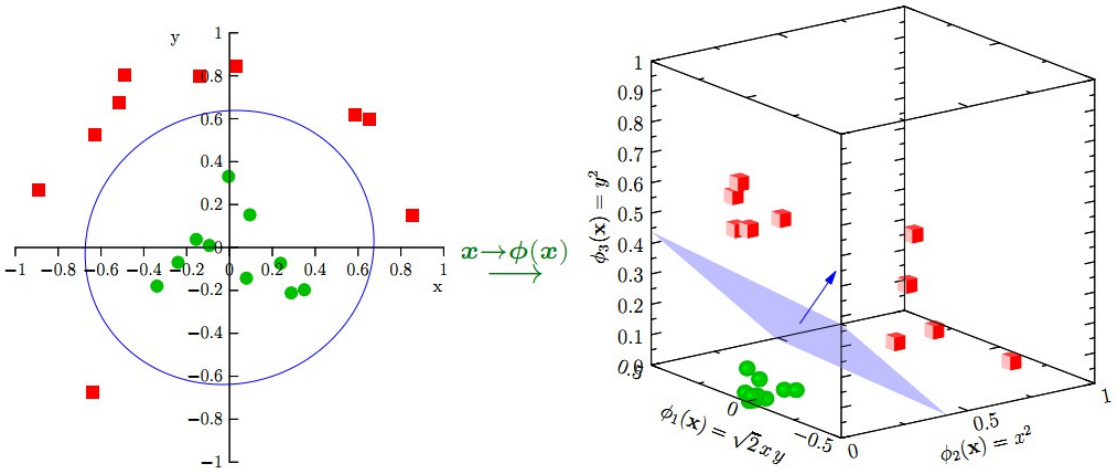
Modern learning machine of choice?

SVMs classify linearly separable data

Finds the maximal stable separating plane



Can increase the likelihood of finding a separating plane by projecting into high dimensional space using the Kernel trick (see later)



SVMs have the ability to separate almost any data set
However, by choosing the largest margin it selects the most robust solution
Thus giving a good generalisation performance despite having a large capacity to learn data
This is termed *structural risk minimisation*

b) Maximal Margins

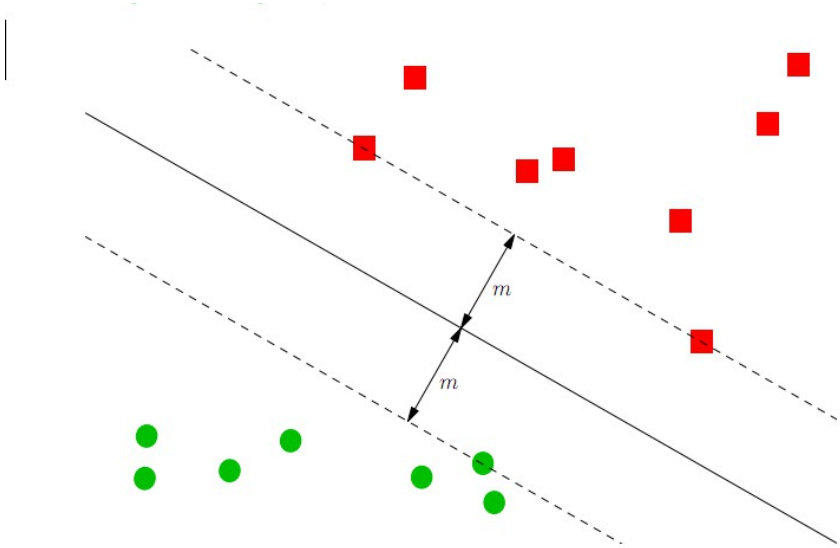
Consider a linearly separable data set

$\{(x_k, y_k)\}_{k=1}^{P}$ and $y_k \in \{-1, 1\}$

Task is to find the vector of weights *w* and bias *b* such that

$y_k(\frac{w^T x_k}{||w||} - b) \geq m$ where *m* is the margin

In the image below, we want to maximise the distance from the central separating plane to the nearest data points
The vector from the line to these points is known as *Support Vectors* and these need to have their length maximise



c) Optimisation Problem

Want to find the *w* and *b* which maximise *m* subject to the constraints

$$y_k(\frac{w^T x_k}{||w||}-b)\geq m \;\forall\, k=1\ldots P$$

Dividing through by *m*, defining $w'=\frac{w}{m||w||}$ and $b'=\frac{b}{m}$ we get the quadratic programming problem:

$$\min_{w',b'}\frac{||w'||^2}{2} \; subject to \; y_k(w'^T x_k-b')\geq 1 \;\forall\, k=1\ldots P$$

We can write the Lagrangian

$$L(w',b',\alpha)=\frac{1}{2}||w'||^2-\sum_{k=1}^{P}\alpha_k(y_k(w'^T x_k-b')-1)$$

$$\nabla_{w'} L=w'-\sum_{k=1}^{P}\alpha_k y_k x_k=0 \quad \text{implying} \quad w_{opt}=\sum_{k=1}^{P}\alpha_k y_k x_k=0$$

$$\frac{\delta L}{\delta b'}=-\sum_{k=1}^{P}\alpha_k y_k=0 \quad \text{implies} \quad \sum_{k=1}^{P}\alpha_k y_k=0$$

Substituting w_{opt} in the condition becomes:

$$\max_{\alpha}\sum_{k=1}^{P}\alpha_k-\frac{1}{2}\sum_{k,l=1}^{P}\alpha_k\alpha_l y_k y_l x_k^T x_l \quad \text{subject to} \quad \sum_{k=1}^{P}\alpha_k y_k=0 \quad \text{and} \quad \alpha_k\geq 0 \;\forall\, k=1..P$$

This is also a quadratic programming problem

However its a *dual form* and depends on the number of training examples (y's) rather than the number of dimensions being searched

This is because the $x_k^T x_l$ is a dot product and just a number

It can be replaced by a Kernel if a search in higher dimensional space is required

$$\max_{\alpha}\sum_{k=1}^{P}\alpha_k-\frac{1}{2}\sum_{k,l=1}^{P}\alpha_k\alpha_l y_k y_l K(x_k,x_l)$$

16 Kernel Trick

a) Kernels

Functions of two variables that can be factorised

$$K(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{y})$$

Where $\phi(\boldsymbol{x})^T = (\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \ldots)$ and $\phi_n(\boldsymbol{x})$ are real valued functions

Factorisation isn't always obvious. More on that later

Rather than working with our inputs, we can work in a transformed feature space which is extended

However, we don't have to explicitly calculate this transformed/higher feature space due to the kernel factorisation

$$\max_{\boldsymbol{\alpha}} \sum_{k=1}^{P} \alpha_k - \frac{1}{2} \sum_{k,l=1}^{P} \alpha_k \alpha_l y_k y_l K(x_k, x_l)$$

That is the kernel trick

b) Factorisation

Sometimes using the factored kernel is easier:

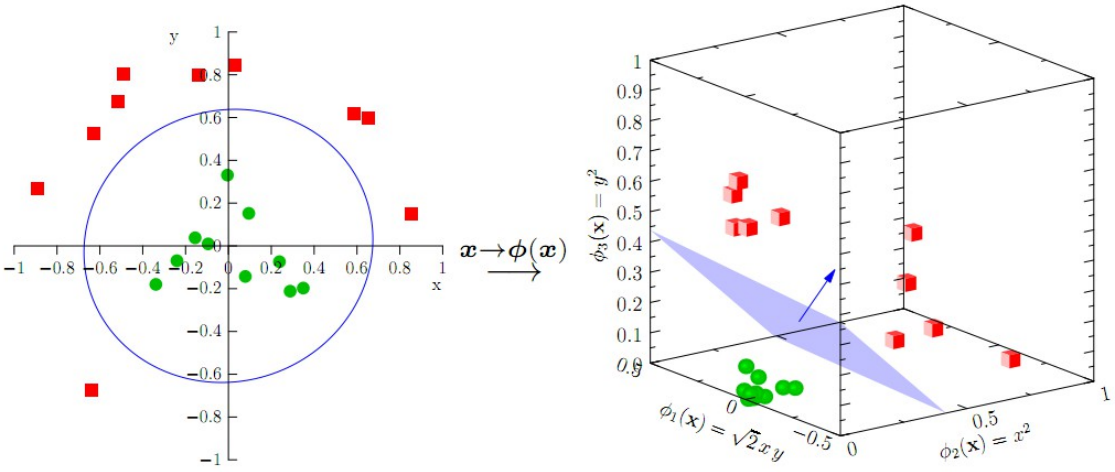
Let $\phi(\boldsymbol{x}) = (x_1^2, x_2^2, \sqrt{2} x_1 x_2)$

Kernel is $K(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{y})$

$$K(\boldsymbol{x}, \boldsymbol{y}) = x_1^2 y_1^2 + x_2^2 y_2^2 + 2 x_1 x_2 y_1 y_2$$

$$K(\boldsymbol{x}, \boldsymbol{y}) = (x_1 y_1 + x_2 y_2)^2$$

$$K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}^T \boldsymbol{y})^2$$



In 2D this kernel projects the data into a 3D space and allows for a measure of distance to be found in the 2D space

c) Eigenfunctions and Kernels

Any function can be made from the sum of its eigen functions and values

Mercers theorem:

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \lambda_i \psi_i(\boldsymbol{x}) \psi_i(\boldsymbol{y})$$

Let $\phi_i(\boldsymbol{x}) = \sqrt{\lambda_i} \psi_i(\boldsymbol{x})$ so $K(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{y})$ but for $\phi_i(\boldsymbol{x})$ to be real valued,

$\lambda_i \geq 0 \forall i$ otherwise the 'distance' between points in the extended space can be negative

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{y})$$

This is the same as saying:

$$K(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{y})$$

So if you like a kernel is the sum of multiple eigenfunctions and their eigenvalues

For SVMs to work the kernels must be projecting into euclidean space so that distances are always positive

Therefore we must use Positive Semi-definite Kernels ($\lambda_i \geq 0 \forall i$) which can always be decomposed into as sum of positive functions

$$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{y})$$

d) Integratal of Kernels

$K(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{y})$ And is positive semi-definite

$$\int \int f(x) K(x, y) f(y) \, dx \, dy \geq 0$$

$$\int \int f(x) \sum_i \phi_i(\boldsymbol{x}) \phi_i(\boldsymbol{y}) f(y) \, dx \, dy$$

$$\sum_i \int f(x) \phi_i(\boldsymbol{x}) \, dx \int \phi_i(\boldsymbol{y}) f(y) \, dy$$

$$\int f(x) \phi_i(\boldsymbol{x}) \, dx = \int \phi_i(\boldsymbol{y}) f(y) \, dy$$ therefore

$$\sum_i \left(\int f(x) \phi_i(\boldsymbol{x}) \, dx \right)^2 = \sum_i \left(\int \phi_i(\boldsymbol{y}) f(y) \, dy \right)^2 \geq 0$$

e) Combining Kernels

Adding

$K_1(x, y)$ and $K_2(x, y)$ are valid kernels

So is $K_3(x, y) = K_1(x, y) + K_2(x, y)$

Multiplied

$K_3(x, y) = K_1(x, y) * K_2(x, y)$ is valid

$K_2(x, y) = K_1(x, y)^n$ is valid too

Exponential

$K_2(x, y) = e^{K_1(x, y)}$ is valid

An important kernel is :

$$K(x, y) = e^{\frac{-\|x - y\|^2}{2}}$$

As it a an infinite number of eigenvalues so if you like its projecting the data into an infinite dimensional space. This is because *e* is an irrational number which can be estimated by an infinite sequence of sums

17 Bayesian Approach

a) Probablistic Reasoning

Probability provides a consistent framework for performing inference (inverse problems)
Many of the best Machine Learning techniques have been developed using this framework
Unfortunately involves ALOT of mathematics

b) Maths

Elementary events
probability used to encode uncertainty about the outcome of events
each outcome is called an elementary event
set of elementary events labelled Ω
For a flip of a coin the elementary events (possible outcomes) are $\Omega=\{heads, tails\}$
Event
An event A is a subset of the elementary events i.e. a set of outcomes
so $A=\{heads\}$ is the event of flipping a heads
The probability of an event occurring is $0\leq P(A)\leq 1$
In a fair coin we would expect $P(\{head\})=P(\{tail\})=\frac{1}{2}$
Rule 1: if we consider a set of exhaustive and mutually exclusive events A_i (for all possible outcomes) then $\sum_i P(A_i)=1$ i.e. the probability of flipping a heads or a tails is 1
Joint Probabilities
If we have two events A and B we can define the joint probability of them both occurring
$P(A, B)$
$P(A)=P(A, B)+ P(A, \neg B)$
if there is a set of exhaustive and mutually exclusive events B_i then
$P(A)=\sum_{(i)} P(A, B_i)$ aka marginalising
Conditional Probabilities
This is the likelihood of an event A occuring given that B has occurred is denoted $P(A \mid B)$
$P(A, B)=P(A \mid B)P(B)=P(B \mid A)P(A)$
$P(A \mid B)=\frac{P(A, B)}{P(B)}=\frac{P(B \mid A)P(A)}{P(B)}$
THIS SAYS NOTHING ABOUT CAUSALITY

This is the second fundamental rule for manipulating probabilities
Independence
If two events A and B are independent then
$P(A, B)=P(A)P(B)$
so $P(A \mid B)=P(A)$
this allows us to ignore events and focus on the ones we're interested in

c) Bayes Rule

Want to decide between different hypothesis $H_1, H_2 .... H_n$
To make the decision we generate some data D

$$P(H \mid D) = \frac{P(D \mid H) P(H)}{P(D)}$$

i.e. the probability of the hypothesis occurring given the data is the probability of the data and hypothesis occurring over the probability of the data occurring

$P(H \mid D)$ is the posterior probability (the probability of H after we know the data)
$P(D \mid H)$ is the likelihood of the data given the hypothesis's
$P(H)$ is the prior probability (before we know the data)
$P(D)$ is the probability of the evidence

We want to know $P(H \mid D)$ i.e. the probability of what happened given some evidence
Bayesian converts this to a forward problem $P(D \mid H)$ where we calculate the likelihood of the data using that hypothesis but we need to know a prior $P(H)$

$$P(D) = \sum_i P(D, H_i) = \sum_i P(D \mid H_i) P(H_i)$$

Example
Have a machine that produced binary digits with a given probability. We are given a sequence of data and want to work out which bias there is.

$D = [0,0,1,0,1]$
$H_1 = 0.2 \mid H_2 = 0.6$ the hypothesis of the probability of getting a 0 bit
$P(H_1) = \frac{1}{2} \mid P(H_2) \frac{1}{2}$ equal probability of each hypothesis
$P(D \mid H_1) = 0.2 * 0.2 * 0.8 * 0.2 * 0.1 = 0.00064$
$P(D \mid H_2) = 0.6 * 0.6 * 0.4 * 0.6 * 0.4 = 0.03456$
$P(D) = P(D \mid H_1) P(H_1) + P(D \mid H_2) P(H_2) = 0.00064 * 0.5 + 0.03456 * 0.5 = 0.0176$
$P(H_1 \mid D) = \frac{P(D \mid H_1) P(H_1)}{P(D)} = \frac{0.00064 * 0.5}{0.0176} = 0.01818$
$P(H_2 \mid D) = \frac{P(D \mid H_2) P(H_2)}{P(D)} = \frac{0.03456 * 0.5}{0.0176} = 0.98181$

As can be seen H2 is much more likely to be true than H1 so the bias of the machine is 0.6 for a 0 being produced

d) Using densities

Probability densities can be used with this model too
This is where the probability of an event *p* taking a value *x* is

$P(p = x) = f(x)$ where *f(x)* is a density function
$P(p = x \mid D) = f(x \mid D)$

$$f(x \mid D) = \frac{P(D \mid x) f(x)}{P(D)}$$

The probability of the data (evidence) is calculated by integrating over the whole probability function

$$P(D) = \int_0^1 P(D \mid p = x) P(p = x) dx = \int_0^1 P(D \mid p = x) f(x) dx$$

e) Chaining

The posterior of one calculation can be used as the prior of another
i.e.

$$f(x \mid X_1) = \frac{P(X_1 \mid x) f(x)}{P(X_1)}$$

$$f(x \mid X_1, X_2) = \frac{P(X_1, X_2 \mid x) f(x \mid X_1)}{P(X_1, X_2)}$$

…

$$f(x \mid X_1 \ldots X_n) = \frac{P(X_1 .. X_n \mid x) f(x \mid X_1 .. X_{n-1})}{P(X_1 \ldots X_n)}$$

f) MAP

Sometimes the full Bayes formula can be hard to use
Therefore Maximum a Posteriori techniques try to find the hypothesis which maximises the posterior
This involves maximising the likelihood and the prior

g) Maximum Likelihood

Let the prior be uniform then we want to maximise the likelihood
i.e. $H = \underset{h}{argmax}\, P(D \mid h)$

If there are a set of independent observations : $D = (X_1, X_2, \ldots, X_n)$ then

$$H = \underset{h}{argmax} \prod_{i=1}^{n} P(X_i \mid h)$$

h) Full Map

Bayesian statistics tells us to consider the prior too therefore full MAP uses it:

$$H = \underset{h}{argmax}\, \Pi \prod_{i=1}^{n} P(X_i \mid h) P(h)$$

This can be used as a regularisation term especially when there is little data and many parameters to be learnt

However, MAP is weaker than Bayes

- throws away lots of informaton
- can't calculate error in prediction
- can't use it in decision theory
- doesn't show clearly whether the hypothesis is much more likely than others (single spike) or is just a local maxima (multiple spikes)

18 Bayes and Machine Learning

a) Application

Bayesian techniques can be used in many different applications
However, they don't automatically fit (like SVMs and MLPs) and so have to be customised to fit the problem

requires thinking and mathematics

Many techniques can be interpreted in a Bayesian way
Take a perceptron:

- we want to learn a hypothesis which is a set of weights *w* from our set of training examples $D=\{(x_k, y_k)\}_{k=0}^{P}$
- we want to know the probability of the weights given the data $P(w|D)$

b) Maximum Likelihood solution

In Bayes $P(w|D) \propto P(D|w)$ so Maximum Likelihood tells us to maximise $P(D|w)$
This is the same as Bayes if $P(w)=const$ as $P(w|D) \propto P(D|w)P(w)$
To maximise the likelihood of the data given some weights, we train on the data and learn a set of weights

Likelihood of data : $P(D|w)=\prod_{k=1}^{P} P((x_k, y_k)|w)$

c) Binary Classification

Sometimes the error function is the mean squared error
Consider a classification system with output y =0 or y = 1
Suppose we want to learn a probability function

$f(x|w)=P(y=1|x)$

Then the likelihood of data $P((x_k, y_k)|w)=\begin{matrix} f(x_k|w) if\ y_k=1 \\ 1-f(x_k|w) if\ y_k=0 \end{matrix}$ this can be expressed as

$f(x_k|w)^{y_k}(1-f(x_k|w))^{1-y_k}$

…

d) Bayesian Learning Machines

Bayes rule used as the basis for many state of the art learning machines

- naïve bayes classifier
- belief networks/ graphical models
- Gaussian processes
- Markov Chains and Monte Carlo methods

They all encode prior knowledge and use a likelihood function to estimate the posterior

e) Naive Bayes Classifier

Used in text classification with good performance
i.e. spam filter
we have a message consisting of words $\langle w_1, w_2 \ldots w_n \rangle$

$$P(Spam|\langle w_1 .. w_n \rangle)=\frac{P(\langle w_1 .. w_n \rangle|Spam)P(Spam)}{P(\langle w_1 .. w_n \rangle)}$$

where $P(\langle w_1 .. w_n \rangle) = P(\langle w_1 .. w_n \rangle | Spam) + P(\langle w_1 .. w_n \rangle | \neg Spam)$

$P(Spam)$ can be estimated using statistics

However, calculating $P(\langle w_1 .. w_n \rangle | Spam)$ would require estimating that every possible message is spam and there and infinite combination of messages.

Naïve Bayes makes an independence assumption:

$$P(\langle w_1 .. w_n \rangle | Spam) = \prod_{k=1}^{n} P(w_i | Spam)$$

i.e. worked out on a word by word basis

$P(w_i | Spam)$ is easier to estimate

However, for rare words this can be mis-leading so some regularisation is needed