# Linear Regression

## Categories of Machine Learning

### Supervised and Unsupervised Learning

- Supervised learning
    - Output class is given in the training data, so the model can determine its own accuracy
- Unsupervised learning
    - No output class is given in the training data, the model tries to make sense by extracting features and patterns on its own

### Machine Learning Models

- Geometric models which use geometry to predict values
    - Separating hyperplanes
    - Linear transformations
    - Distance metrics
- Probabilistic models which use probability distributions and methods of reducing uncertainty to create models
    - Naive bayes
    - Big advantage is that we can say how certain we are of the prediction
- Regression models which have the goal of predicting a numeric output
    - Linear regression
- Classification models which predict discrete values, enumerated types
    - Tree learning
    - KNN
- Neural networks which learn based on a biological analogy of perceptrons
- Local models which predict in the local region of query instances
    - KNN
- Tree-based models which partition data to make predictions
    - Decision trees
- Ensembles

### Linear Models

- Numeric attributes and numeric prediction, i.e., regression
- Linear models, i.e. outcome is *linear* combination of attributes

$$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_n x_n$$

-

- Weights are calculated from the training data
- **Predicted** value for first training instance x(1) is:

$$b_0 x_0^{(1)} + b_1 x_1^{(1)} + b_2 x_2^{(1)} + \ldots + b_n x_n^{(1)} = \sum_{i=0}^{n} b_i x_i^{(1)}$$

-

## Squared Error

In order to find the best model for linear regression we want to find one such that the difference between the predicted and actual values across the entire model is minimised.

## Probability vs Statistics

- Probability reasons from populations to samples
    - Deductive reasoning
    - Sound reasoning
- Statistics reasons from samples to populations
    - Inductive reasoning
    - Unsound reasoning
    - Usually requires a large amount of data to make a conclusion with any backbone

## Sampling and extracting Data

- For homogenous groups we don't need to collect a lot of data
    - You don't need to touch the burning stove more than once to decide its hot
- For groups which have irregularities, we need to take measurements from the entire group in order to have meaningful data
- Sampling is a technique to draw conclusions about a group without having to measure all of the population
    - The conclusions don't need to be completely accurate
- This is all based on the assumption that the sample closely represents the group we are trying to make a conclusion about.
- We want to make sure that there is no systematic bias or any bias that we cannot account for in our calculations
- The chance of obtaining a sample that is unrepresentative can be calculated and based on this calculation we can determine if we want to draw any conclusion or base how confident we are in our conclusion
- The larger the sample, the lower the chance we get a sample that is unrepresentative

When we need a model to predict an output numeric value, linear regression is the way to go. Linear regression tries to fit a line equation that can be used to predict values on new input values. We attempt to predict the coefficients of each feature such that we minimise the

difference between the predicted values and the actual values. This is known as OLS (ordinary least square regression).

## Inductive Learning Hypothesis

Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

## Estimation from a Sample

- Estimating some aspect of the population using a sample is a common task. Along with the estimate, we also want to have some idea of the accuracy of the estimate (usually expressed in terms of confidence limits)
- Some measures calculated from the sample are very good estimates of corresponding population values. For example, the sample mean m is a very good estimate of the population mean μ. But this is not always the case. For example, the range of a sample usually under-estimates the range of the population
- We will have to clarify what is meant by a “good estimate”. One meaning is that an estimator is correct on average. For example, on average, the mean of a sample is a good estimator of the mean of the population
- For example, when a number of samples are drawn and the mean of each is found, then average of these means is equal to the population mean
- Such an estimator is said to be statistically unbiased

## Mean

This is calculated as follows:

- Find the total T of N observations. Estimate the (arithmetic) mean by m=T/N
- This works very well when the data follow a symmetric bell-shaped frequency distribution (of the kind modelled by “normal” distribution)
  - A simple mathematical expression of this is $m = \frac{1}{N}\sum_i x_i$, where the observations are $x_1$, $x_2$ $\ldots x_n$
  - If we can group the data so that the observation $x_1$ occurs $f_1$ times, $x_2$ occurs $f_2$ times and so on, then the mean is calculated as $m = \frac{1}{\sum_i f_i}\sum_i x_i f_i$
-

- If, instead of frequencies, you had relative frequencies (i.e. instead of $f_i$ you had $p_i = f_i/N$), then the mean is simply the observations weighted by relative frequency. That is, $m = \sum_i x_i p_i$

- We want to connect this up to computing the mean value of observations modelled by some theoretical probability distribution function. That is, we want to asimilar counting method for calculating the mean of random variables modelled using some known distribution
- Correctly, this is the mean value of the values of the random variable function. But this is a bit cumbersome,so we will just say the "mean value of the r.v." For discrete r.v.'s this is:

$$E(X) = \sum_i x_i p(X = x_i)$$

## Variance

.This is calculated as follows:

- Calculate the total $T$ and the sum of squares of $N$ observations. The estimate of the standard deviation is $s = \sqrt{\frac{1}{N-1}\sum_i (x_i - m)^2}$
- Again, this is a very good estimate when the data are modelled by a normal distribution

- For grouped data, this is modified to $s = \sqrt{\frac{1}{N-1}\sum_i (x_i - m)^2 f_i}$
- Again, we have a similar formula in terms of expected values, for the scatter (spread) of values of a r.v. $X$ around a mean value $E(X)$:

$$\begin{aligned} Var(X) &= E((X - E(X))^2) \\ &= E(X^2) - [E(X)]^2 \end{aligned}$$

- You can remember this as "the mean of the squares minus the square of the mean"

## Correlation

The correlation coefficient is a number between -1 and +1 that indicates whether a pair of variables x and y are associated, and the scatter in association.

- Higher values of x are associated with higher values of y and low values of x are associated with low values of y implies that scatter is low
- A value close to 0 indicates no association and large scatter associated with values
- A value close to -1 suggests an inverse association between x and y

Note this is only appropriate and accurate when x and y are associated linearly, it doesn't work well for curves or higher order polynomials (e.g. y=x^2)

The formula for computing correlation between x and y is the following

$$r = \frac{\text{cov}(x, y)}{\sqrt{\text{var}(x)}\sqrt{\text{var}(y)}}$$

Since the denominator is just the standard deviations of x and y, we can rewrite the calculator as the deviations from the mean of the values, using the formula for covariance

$$\text{cov}(x, y) = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{n - 1}$$

- If the positive products dominate in the summation then we can say the value of r will be positive
- If the negative products dominate in the summation we can say the value of r will be negative
- If the products are all similar in the negative and positive, we can say the value of r will be close to 0

By substituting the covariance into our equation and using the equation for variance we get

- You should be able to show that:

$$r = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum (x_i - \overline{x})^2}\sqrt{\sum (y_i - \overline{y})^2}}$$

- Computers generally use a short-cut formula:

$$r = \frac{\sum_i x_i y_i \; - \; n\overline{xy}}{n - 1}$$

We can use this value to quickly check whether there is some linear association between x and y. The sign of the value tells the direction of the association, and all the value tells you is the scatter.

It is important to note the correlation is not used for any form of modelling and cannot tell us values of y given values of x. It is very possible for two datasets to have the same correlation but different relationships, and it is also possible for two datasets to have the same relationship but different correlation.

Correlation does not mean causation, and it cannot be used to compare datasets at all, all we can tell is the association between x and y.

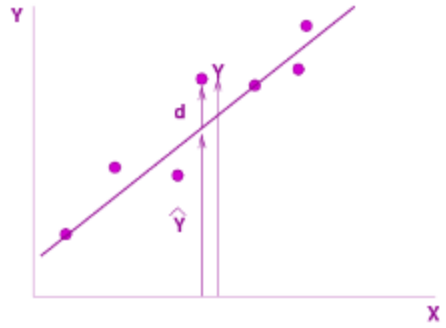
## Regression

- Given a set of data points $x_i, y_i$, what is the relationship between them? (We can generalise this to the "multivariate" case later)
- One kind of question is to ask: are these linearly related in some manner? That is, can we draw a straight line that describes reasonably well the relationship between $X$ and $Y$
- Remember, the correlation coefficient can tell us if there is a case for such a relationship
- In real life, even if such a relationship held, it will be unreasonable to expect all pairs $x_i, y_i$ to lie precisely on a straight line. Instead, we can probably draw some reasonably well-fitting line. But which one?

-

# Univariate Linear Regression

## Linear Relationship Between Two Variables




- GOAL: fit a line whose equation is of the form $\hat{y} = a + bx$
- HOW: minimise $\sum_i d_i^2 = \sum_i (y_i - \hat{y}_i)^2$ (the "least squares estimator")

- The calculation for $b$ is given by:

$$b = \frac{\text{cov}(x, y)}{\text{var}(x)}$$

  where $\text{cov}(x, y)$ is the covariance of $x$ and $y$, given by $\sum_i (x_i - \overline{x})(y_i - \overline{y})$ as before
- Then we have $a = \overline{y} - b\overline{x}$

## Meaning of the Coefficients a and b

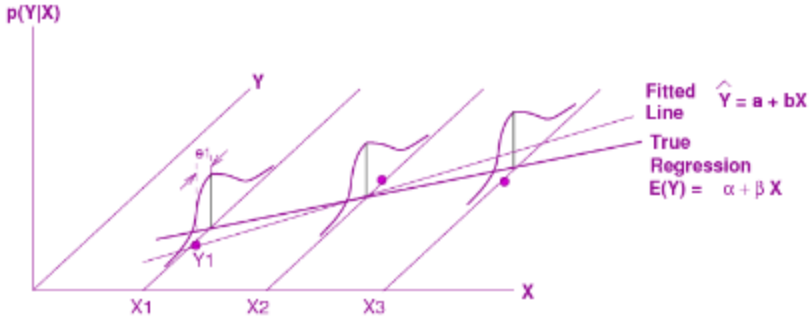
- b: change in Y that accompanies a unit change in X
- If the values of X were assigned at random, then b estimates the unit change in Y caused by a unit change in X

- If the values of X were not assigned at random (for example, they were data somebody observed), then the change in Y will include the change in X and any other confounding variables that may have changed as a result of changing X by 1 unit. So, you cannot say for example, that a change of X by 1 unit causes b units of change in Y
- b = 0 means there is no linear relationship between X and Y, and then best we can do is simply say is $\hat{Y} = a = \overline{Y}$. Estimating the sample mean is therefore a special case of the MSE criterion

## The Regression Model

- The least-squares estimator fits a line using sample data
- To draw inferences about the population requires us to have a (statistical) model about what this line means

What is being assumed is actually this:




This diagram tells us that throughout any sample point, the deviation from the true regression line stays the same. We don't want the distribution of error to get wider as we increase our value of x. In this example, we don't want the value at x1 to be closer to the regression line than the value at x3.

- That is: Obtain $Y$ values for many instances of $X_1$. This will result in a distribution of $Y$ values $P(Y|X_1)$; and so on for $P(Y|X_2), P(Y|X_3), etc.$. The regression model makes the following assumptions:
  - All the $Y$ distributions are the same, and have the same spread
  - For each $P(Y|X_i)$ distribution, the true mean value $\mu_i$ lies on a straight line (this is the "true regression line")
  - The $Y_i$ are independent
- In standard terminology, the $Y_i$ are independent and identically distributed (i.i.d.) random variables with mean $\mu_i = \alpha + \beta X_i$ and variance $\sigma^2$
- Or: $Y_i = \alpha + \beta X_i + e_i$ where the $e_i$ are independent errors with mean $0$ and variance $\sigma^2$

## Finding Parameters

Univariate linear regression assumes a linear equation w = a + bh, where parameters a and b are chosen to minimise the least squared error.

$$\sum_{i=1}^{n}(w_i - (a + bh_i))^2$$

In order to find these parameters we take partial derivatives and solve for when they are equal to 0.

$$\frac{\partial}{\partial a}\sum_{i=1}^{n}(w_i - (a + bh_i))^2 = -2\sum_{i=1}^{n}(w_i - (a + bh_i)) = 0$$
$$\Rightarrow \hat{a} = \overline{w} - \hat{b}\overline{h}$$

$$\frac{\partial}{\partial b}\sum_{i=1}^{n}(w_i - (a + bh_i))^2 = -2\sum_{i=1}^{n}(w_i - (a + bh_i))h_i = 0$$
$$\Rightarrow \hat{b} = \frac{\sum_{i=1}^{n}(h_i - \overline{h})(w_i - \overline{w})}{\sum_{i=1}^{n}(h_i - \overline{h})^2}$$

So the solution found by linear regression is $w = \hat{a} + \hat{b}h = \overline{w} + \hat{b}(h - \overline{h})$.

For a feature X and target variable Y the regression coefficient (in univariate linear regression) is the covariance between X and Y in proportion to eh variance in X

$$\hat{b} = \frac{\sigma_{XY}}{\sigma_{XX}}$$

The intercept can be calculated using a given point in the equation (mean X, mean Y) and all our known variables. Adding a translation will affect only the intercept but not the regression coefficient.

We can also note by solving in the above way we get the following

.

$$\sum_{i=1}^{n}(y_i - (\hat{a} + \hat{b}x_i)) = n(\overline{y} - \hat{a} - \hat{b}\overline{x}) = 0$$

The result follows because $\hat{a} = \overline{Y} - \hat{b}\overline{X}$, as derived above.

### Outliers

Whilst it is appealing to have our sum of least squared errors 0, it does leave this susceptible to outliers, this can have a serious effect on our model, and this also means we need to make sure we get rid of outliers during data collection.

## Multivariate Linear Regression

Similar to univariate linear regression however, our equation line takes the following form

- The $Y_i$ are identically distributed independent variables with mean $\mu = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n$ and variance $\sigma^2$
- Or: $Y_i = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n + e_i$ where the $e_i$ are independent errors with mean $0$ and variance $\sigma^2$
- As before, this linear model is estimated from a sample by the equation $\hat{Y} = b_0 + b_1 X_1 + b_2 X_2 + \cdots + b_n X_n$

### Regularisation

To avoid overfitting our solution to our training set we make sure that the weights on average are small in magnitude. We can do this by adding penalty terms to the cost function which forces coefficients to shrink to zero.

- MSE as a cost function, given data $(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m)$

$$Cost(\theta) = \frac{1}{m}\sum^{m}_{i=1}(f_\theta(\mathbf{x_i}) - y_i)^2$$

and with a penalty function:

$$Cost(\theta) = \frac{1}{m}\sum^{m}_{i=1}(f_\theta(\mathbf{x_i}) - y_i)^2 + \frac{1}{m}\lambda\sum^{m}_{i=1}\theta_i$$

Using the second version of estimating cost, we make sure our parameters don't get too large. The first part is the MSE however, we sum all the coefficients of our feature parameters, and

make sure that our coefficients aren't too large. This will greatly help with over-fitting issues and is known as regularisation. Large coefficients make the function more non-linear and over-fit to adjust for errors. This more advanced cost function now allows us to control the tradeoff between complexity of our estimation and the accuracy of our solution, by the use of the lambda function, if we want a more accurate solution with less errors on training we take a smaller lambda, when we want a more general function that does not rely so heavily on training data, but suffers potentially from accuracy, a higher lambda will help.

Using Lasso regression, we eliminate useless features and only keep features which have impact.

An interesting alternative form of regularised regression is provided by the *lasso*, which stands for 'least absolute shrinkage and selection operator'. It replaces the ridge regularisation term $\sum_i w_i^2$ with the sum of absolute weights $\sum_i |w_i|$. The result is that some weights are shrunk, but others are set to 0, and so the lasso regression favours *sparse solutions*.

## Gradient Descent

A technique for finding the parameters needed is by using gradient descent with the penalty function. To do this we will

1. Move each feature coefficients in a direction that minimises the cost and
2. Each value of our coefficients will get shrunk on each iteration by multiplying the old value by an amount smaller than 1

$$\theta_j^{(i+1)} = \alpha\theta_j^{(i)} - \eta\nabla_{\theta_j}$$

where $\alpha < 1$.
Note the slight change of notation above: now $j$ is being used to index the parameters $\theta_j$, while $i$, $i+1$ denote iterations of the gradient descent procedure.

The multivariate least-squares regression problem can be written as an optimisation problem:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\mathbf{w})$$

The regularised version of this optimisation is then as follows:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda||\mathbf{w}||^2$$

where $||\mathbf{w}||^2 = \sum_i w_i^2$ is the squared norm of the vector $\mathbf{w}$, or, equivalently, the dot product $\mathbf{w}^{\mathrm{T}}\mathbf{w}$; $\lambda$ is a scalar determining the amount of regularisation.

Note that w = weights or coefficients on the features

$$\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

# Tutorial

## Partial Derivative

Obtain the partial derivatives with respect to one value, when

$$f(x, y) = a_1x^2y^2 + a_4xy + a_5x + a_7$$

---

**Answer**

$$\frac{\partial f(x,y)}{\partial x} = 2x(a_1y^2) + a_4y + a_5$$

$$\frac{\partial f(x,y)}{\partial y} = 2y(a_1x^2) + a_4x$$

---

Q2

When

$$f(x,y) = a_1x^2y^2 + a_2x^2y + a_3xy^2 + a_4xy + a_5x + a_6y + a_7$$

what will $\frac{\partial f(x,y)}{\partial x}$ and $\frac{\partial f(x,y)}{\partial y}$ be?

---

**Answer**

$$\frac{\partial f(x,y)}{\partial x} = 2a_1xy^2 + 2a_2xy + a_3y^2 + a_4x + a_5$$

$$\frac{\partial f(x,y)}{\partial x} = 2a_1x^2y + a_2x^2 + 2a_3xy2 + a_4x + a_6$$

---

## R3

Do you recall how to solve optimization problems for Quadratic function? For example, $y = 4x^2 - 3x + 3$. Is the solution a minimum or maximum?

---

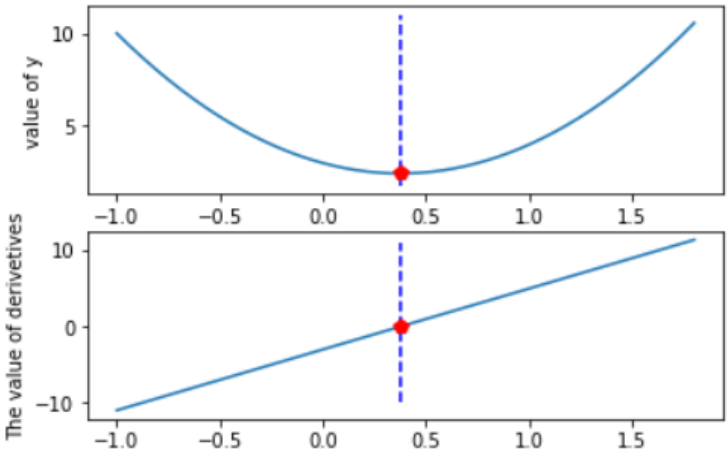
**Answer**

$$\frac{\partial y}{\partial x} = 8x - 3$$

when $x = \frac{3}{8}, \frac{\partial y}{\partial x} = 0$ and y is minimum.

---

Hint:




# Loss Function

## R4

What is the *loss function* for linear regression?

---

**Answer**

$$\mathcal{L}(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2$$

.

---

# Loss Function Partial Derivative

## Q1

***Go through this derivation and complete the exercise at the end of it.***

A *univariate linear regression model* is a linear equation $y = a + bx$. Learning such a model requires fitting it to a sample of training data so as to minimize the error function $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2$. To find the best parameters $a$ and $b$ that minimize this error function we need to find the error *gradients* $\frac{\partial \mathcal{L}}{\partial w_0}$ and $\frac{\partial \mathcal{L}}{\partial w_1}$. So we need to derive these expressions by taking partial derivatives, set them to zero, and solve for $w_0$ and $w_1$.

First we write the loss function for the univariate linear regression $y = w_0 + w_1 x$ as

$$\mathcal{L} \;=\; \mathcal{L}(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - w_0 - w_1 x_i)^2$$

At a minimum of $\mathcal{L}$, the partial derivatives with respect to $w_0$, $w_1$ should be zero. We will start with taking the partial derivative of $\mathcal{L}$ with respect to $w_0$:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}\frac{1}{n}\sum_{i=1}^{n}(y_i - w_0 - w_1 x_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial w_0}(y_i - w_0 - w_1 x_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} -2(y_i - w_0 - w_1 x_i) \\
&= -2\left[\frac{1}{n}\sum_{i=1}^{n} y_i - w_0\frac{1}{n}\sum_{i=1}^{n} 1 - w_1\frac{1}{n}\sum_{i=1}^{n} x_i\right] \\
&= -2\left[\overline{y} - w_0 - w_1\overline{x}\right],
\end{aligned}$$

where we have introduced the notation $\overline{f}$ to mean the sample average of $f$, i.e. $\overline{f} = \frac{1}{m}\sum_{j=1}^{m} f_j$, where $m$ is the length of $f$. Now, we equate this to zero and solve for $w_0$ to get

$$-2\left[\overline{y} - w_0 - w_1\overline{x}\right] \;=\; 0 \implies w_0 = \overline{y} - w_1\overline{x}$$

Note, we have not actually solved for $w_0$ yet, since our expression depends on $w_1$, which we must also optimise over.

Taking the partial derivative of $\mathcal{L}$ with respect to $w_1$:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial}{\partial w_1}\frac{1}{n}\sum_{i=1}^{n}(y_i - w_0 - w_1 x_i)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} -2x_i(y_i - w_0 - w_1 x_i) \\
&= -2\left[\frac{1}{n}\sum_{i=1}^{n} x_i y_i - w_0\frac{1}{n}\sum_{i=1}^{n} x_i - w_1\frac{1}{n}\sum_{i=1}^{n} x_i^2\right] \\
&= -2\left[\overline{xy} - w_0\overline{x} - w_1\overline{x^2}\right],
\end{aligned}$$

Now, we equate this to zero and solve for $w_1$ to get

$$-2\left[\overline{xy} - w_0\overline{x} - w_1\overline{x^2}\right] = 0 \implies w_1 = \frac{\overline{xy} - w_0\overline{x}}{\overline{x^2}}$$

We now have an expression for $w_0$ in terms of $w_1$, and an expression for $w_1$ in terms of $w_0$. These are known as the Normal Equations. In order to get an explicit solution for $w_0, w_1$, we can plug $w_0$ into $w_1$ and solve:

$$\begin{aligned}
w_1 &= \frac{\overline{xy} - w_0\overline{x}}{\overline{x^2}} \\
&= \frac{\overline{xy} - (\overline{y} - w_1\overline{x})\overline{x}}{\overline{x^2}} \\
&= \frac{\overline{xy} - \overline{x}\,\overline{y} + w_1\overline{x}^2}{\overline{x^2}}
\end{aligned}$$

Rearranging and solving for $w_1$ gives us

$$w_1 = \frac{\overline{xy} - \overline{x}\,\overline{y}}{\overline{x^2} - \overline{x}^2}$$

So now we have an explicit solution for the regression parameters $w_0$ and $w_1$, and so we are done.

***Exercise:*** To make sure you know the process, try to solve the following loss function for linear regression with a version of "$L2$" regularization, in which we add a penalty that penalizes the size of $w_1$. Let $\lambda > 0$ and consider the regularised loss

$$\mathcal{L}(w_0, w_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2 + \lambda w_1^2$$

**Answer**

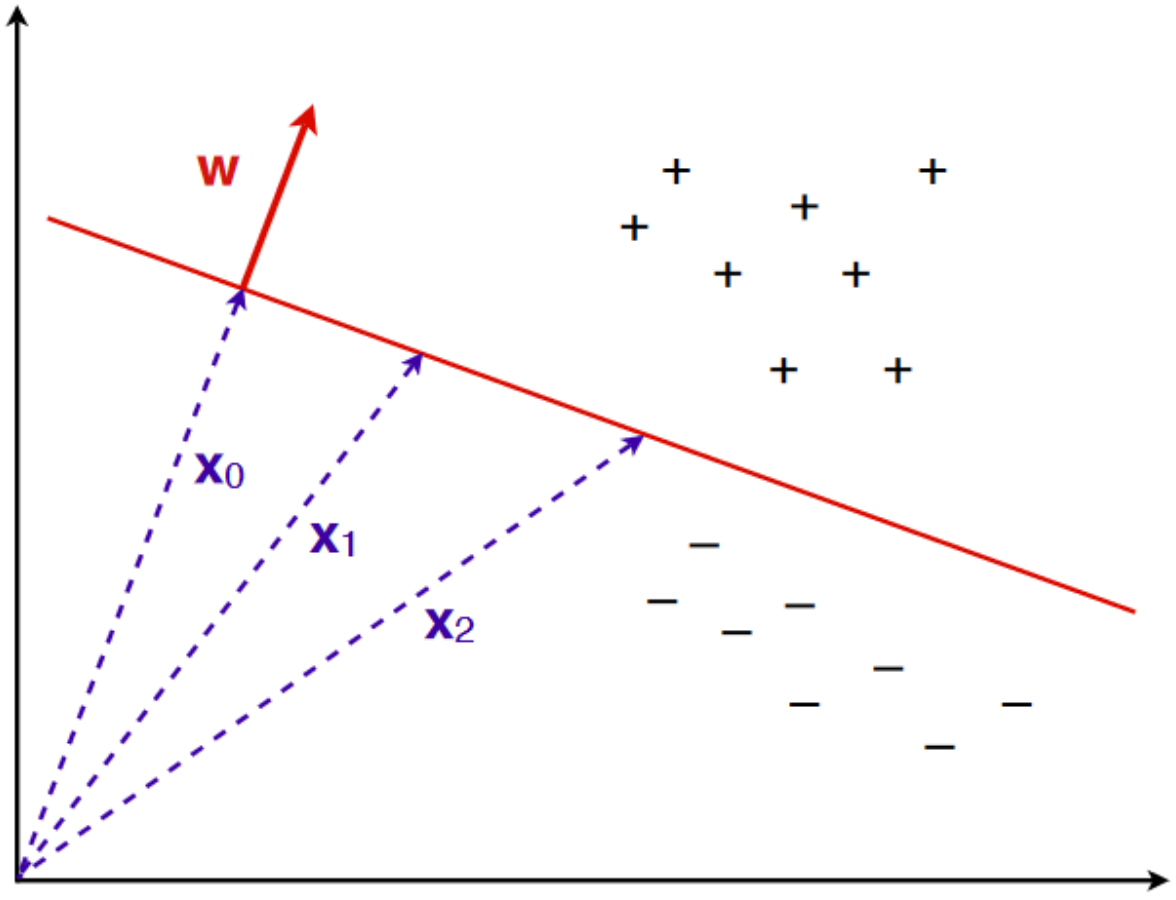
$$\begin{aligned}
w_0 &= \overline{y} - w_1\overline{x} \\
w_1 &= \frac{\overline{xy} - \overline{x}\,\overline{y}}{\overline{x^2} - \overline{x}^2 + \lambda}
\end{aligned}$$

# Classification 1

## Introduction

Classification (sometimes called concept learning) methods dominate machine learning . . .. . . However, they often don't have convenient mathematical properties like regression, so are more complicated to analyse. The idea is to learn a classifier, which is usually a function mapping from an input data point to one of a set of discrete outputs, i.e., the classes.We will mostly focus on their advantages and disadvantages as learning methods first, and point to unifying ideas and approaches where applicable. In this and the next lecture we focus on classification methods that are essentially linear models. . .and in later lectures we will see other, more expressive, classifier learning methods.

## Linear classification in two dimensions




- straight line separates positives from negatives

- defined by w·x_i = t
- W is perpendicular to decision boundary
- W points in direction of positives
- t is the decision threshold

Note: $\mathbf{x}_i$ points to a point on the decision boundary. In particular, $\mathbf{x}_0$ points in the same direction as $\mathbf{w}$, from which it follows that $\mathbf{w} \cdot \mathbf{x}_0 = ||\mathbf{w}||\ ||\mathbf{x}_0|| = t$ (where $||\mathbf{x}||$ denotes the length of the vector $\mathbf{x}$).

## Homogeneous coordinates

It is sometimes convenient to simplify notation further by introducing an extra constant 'variable' $x_0 = 1$, the weight of which is fixed to $w_0 = -t$.

The extended data point is then $\mathbf{x}^\circ = (1, x_1, \ldots, x_n)$ and the extended weight vector is $\mathbf{w}^\circ = (-t, w_1, \ldots, w_n)$, leading to the decision rule $\mathbf{w}^\circ \cdot \mathbf{x}^\circ > 0$ and the decision boundary $\mathbf{w}^\circ \cdot \mathbf{x}^\circ = 0$.

Thanks to these so-called *homogeneous coordinates* the decision boundary passes through the origin of the extended coordinate system, at the expense of needing an additional dimension.

Note: this doesn't really affect the data, as all data points and the 'real' decision boundary live in the plane $x_0 = 1$.

## A Bayesian classifier I

Bayesian spam filters maintain a vocabulary of words and phrases – potential spam or ham indicators – for which statistics are collected from a training set.

- For instance, suppose that the word 'Viagra' occurred in four spame-mails and in one ham e-mail. If we then encounter a new e-mail that contains the word 'Viagra', we might reason that the odds that this e-mail is spam are 4:1, or the probability of it being spam is 0.80 and the probability of it being ham is 0.20.
- The situation is slightly more subtle because we have to take into account the prevalence of spam. Suppose that I receive on average one spam e-mail for every six

ham e-mails. This means that I would estimate the odds of an unseen e-mail being spam as 1:6, i.e.,non-negligible but not very high either.

- If I then learn that the e-mail contains the word ‘Viagra’, which occurs four times as often in spam as in ham, I need to combine these two odds. As we shall see later, Bayes’ rule tells us that we should simply multiply them: 1:6 times 4:1 is 4:6, corresponding to a spam probability of 0.4.

In this way you are combining two independent pieces of evidence, one concerning the prevalence of spam, and the other concerning the occurrence of the word ‘Viagra’, pulling in opposite directions.

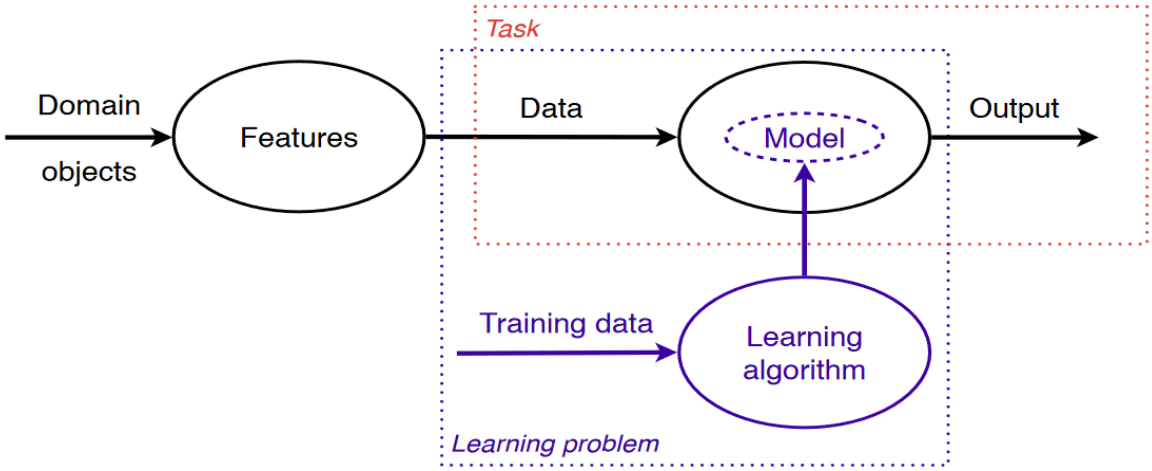
The nice thing about this ‘Bayesian’ classification scheme is that it can be repeated if you have further evidence. For instance, suppose that the odds in favour of spam associated with the phrase ‘blue pill’ is estimated at 3:1,and suppose our e-mail contains both ‘Viagra’ and ‘blue pill’, then the combined odds are 4:1 times 3:1 is 12:1, which is ample to outweigh the 1:6 odds associated with the low prevalence of spam (total odds are 2:1, ora spam probability of 0.67, up from 0.40 without the ‘blue pill’).

## A rule-based classifier

- if the e-mail contains the word ‘Viagra’ then estimate the odds of spam as 4:1;
- otherwise, if it contains the phrase ‘blue pill’ then estimate the odds of spam as 3:1;
- otherwise, estimate the odds of spam as 1:6.

The first rule covers all e-mails containing the word ‘Viagra’, regardless of whether they contain the phrase ‘blue pill’, so no overcounting occurs.The second rule only covers e-mails containing the phrase ‘blue pill’ but not the word ‘Viagra’, by virtue of the ‘otherwise’ clause. The third rule covers all remaining e-mails: those which contain neither ‘Viagra’ nor ‘blue pill’.

## How machine learning helps to solve a task




An overview of how machine learning is used to address a given task. A task (red box) requires an appropriate mapping – a model – from data described by features to outputs. Obtaining such a mapping from training data is what constitutes a learning problem (blue box).

## Some terminology

Tasks are addressed by models, whereas learning problems are solved by learning algorithms that produce models.

Machine learning is concerned with using the right features to build the right models that achieve the right tasks.

Models lend the machine learning field diversity, but tasks and features give it unity.

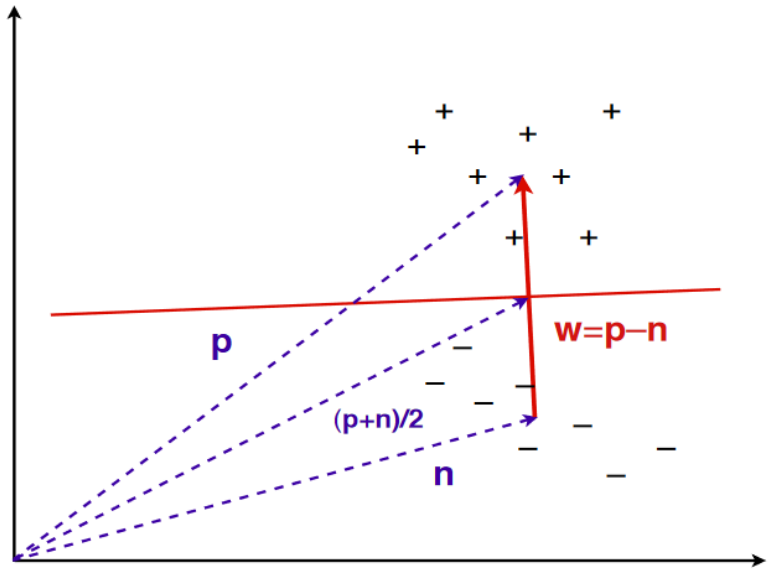
Does the algorithm require all training data to be present before the start of learning ? If yes, then it is categorised as a **batch learning** algorithm.

If however, it can continue to learn when new data arrives, it is an **online learning** algorithm.

If the model has a fixed number of parameters it is categorised as **parametric**.

Otherwise, if the number of parameters grows with the amount of training data it is categorised as **non-parametric**.

## Basic Linear Classifier




The basic linear classifier constructs a decision boundary by half-wayintersecting the line between the positive and negative centres of mass.

The basic linear classifier is described by the equation $\mathbf{w} \cdot \mathbf{x} = t$, with $\mathbf{w} = \mathbf{p} - \mathbf{n}$; the decision threshold can be found by noting that $(\mathbf{p} + \mathbf{n})/2$ is on the decision boundary, and hence $t = (\mathbf{p} - \mathbf{n}) \cdot (\mathbf{p} + \mathbf{n})/2 = (||\mathbf{p}||^2 - ||\mathbf{n}||^2)/2$, where $||\mathbf{x}||$ denotes the length of vector $\mathbf{x}$.

## Deduction vs Induction

**Deduction**: derive specific consequences from general theories
**Induction**: derive general theories from specific observations
Deduction is well-founded (mathematical logic).
Induction is (philosophically) problematic – induction is useful since it often seems to work – an inductive argument!

## Generalisation - the key objective of machine learning

What we are really interested in is generalising from the sample of data in our training set. This can be stated as:

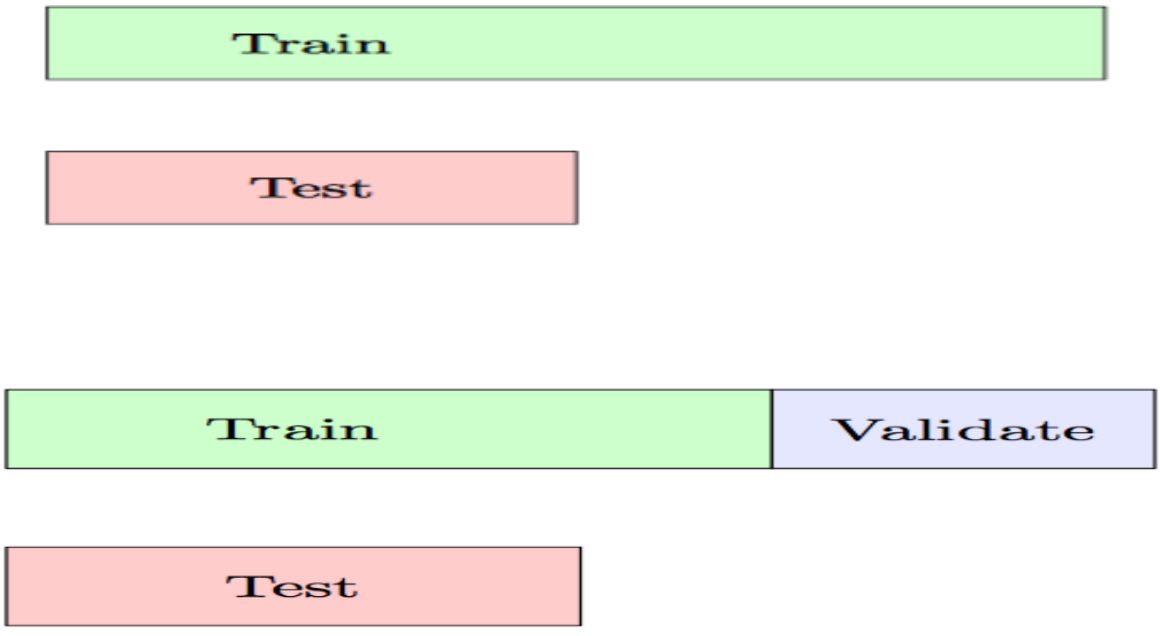
**The inductive learning hypothesis**

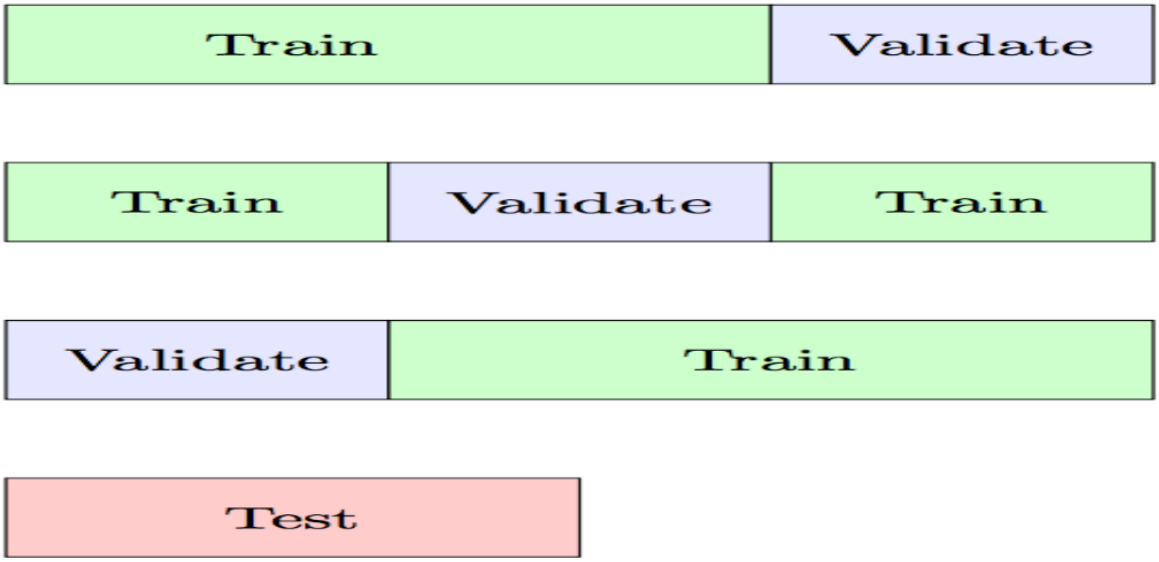
Any hypothesis found to approximate the target (true) functionwell over a sufficiently large set of training examples will also ap-proximate the target function well over other unobserved examples.

A corollary of this is that it is necessary to make some assumptions about the type of target function in a task for an algorithm to go beyond the data, i.e., generalise or learn.

## Cross-validation I

There are certain parameters that need to be estimated during learning.We use the data, but NOT the training set, OR the test set. Instead, we use a separate validation or development set.






## Contingency table

Two-class prediction case:

| Actual Class | Predicted Class: Yes | Predicted Class: No |
|---|---|---|
| Yes | True Positive (TP) | False Negative (FN) |
| No | False Positive (FP) | True Negative (TN) |

Classification Accuracy on a sample of labelled pairs $(x, c(x))$ given a learned classification model that predicts, for each instance $x$, a class value $\hat{c}(x)$:

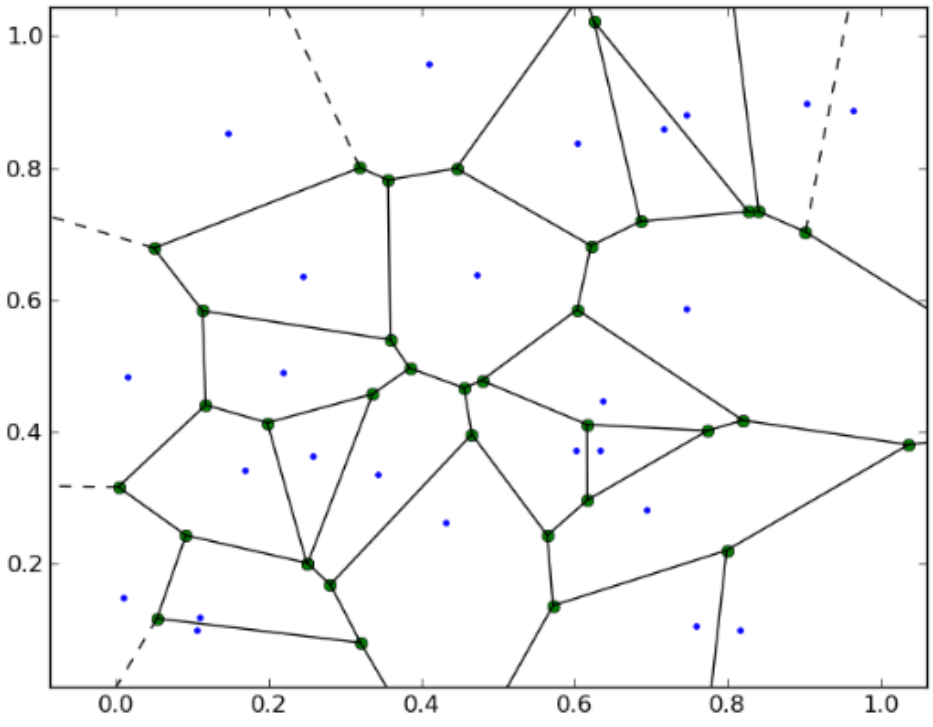
$$\text{acc} = \frac{1}{|\text{Test}|} \sum_{x \in \text{Test}} I[\hat{c}(x) = c(x)]$$

where Test is a test set and $I[]$ is the indicator function which is 1 iff its argument evaluates to true, and 0 otherwise.

Classification Error is 1-acc.

Simply we take how many incorrectly labelled points and divide by the total number of points to get percentage accuracy

## Nearest Neighbour



Nearest Neighbour is a regression or classification algorithm that predicts whatever is the output value of the nearest data point to some query.

## Minkowski distance

*Minkowski distance* If $\mathcal{X} = \mathbb{R}^d$, the *Minkowski distance* of order $p > 0$ is defined as

$$\text{Dis}_p(\mathbf{x}, \mathbf{y}) = \left( \sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p} = ||\mathbf{x} - \mathbf{y}||_p$$

where $||\mathbf{z}||_p = \left( \sum_{j=1}^{d} |z_j|^p \right)^{1/p}$ is the $p$-*norm* (sometimes denoted $L_p$ norm) of the vector $\mathbf{z}$.

- The $2$-norm refers to the familiar *Euclidean distance*

$$\text{Dis}_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2} = \sqrt{(\mathbf{x} - \mathbf{y})^{\mathrm{T}} (\mathbf{x} - \mathbf{y})}$$

  which measures distance 'as the crow flies'.
- The $1$-*norm* denotes *Manhattan distance*, also called *cityblock distance*:

$$\text{Dis}_1(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{d} |x_j - y_j|$$

  This is the distance if we can only travel along coordinate axes.

- If we now let $p$ grow larger, the distance will be more and more dominated by the largest coordinate-wise distance, from which we can infer that $\mathrm{Dis}_\infty(\mathbf{x},\mathbf{y}) = \max_j |x_j - y_j|$; this is also called *Chebyshev distance*.
- You will sometimes see references to the $0$-*norm* (or $L_0$ norm) which counts the number of non-zero elements in a vector. The corresponding distance then counts the number of positions in which vectors $\mathbf{x}$ and $\mathbf{y}$ differ. This is not strictly a Minkowski distance; however, we can define it as

$$\mathrm{Dis}_0(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{d} (x_j - y_j)^0 = \sum_{j=1}^{d} I[x_j = y_j]$$

  under the understanding that $x^0 = 0$ for $x = 0$ and $1$ otherwise.

Sometimes the data $\mathcal{X}$ is not naturally in $\mathbb{R}^d$, but if we can turn it into Boolean features, or character sequesnces, we can still apply distance measures. For example:

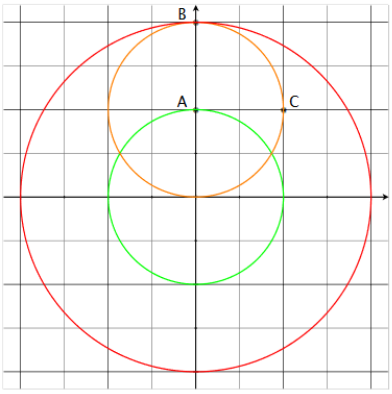
- If $\mathbf{x}$ and $\mathbf{y}$ are binary strings, this is also called the *Hamming distance*. Alternatively, we can see the Hamming distance as the number of bits that need to be flipped to change $\mathbf{x}$ into $\mathbf{y}$.
- For non-binary strings of unequal length this can be generalised to the notion of *edit distance* or *Levenshtein distance*.

## Distance Metric

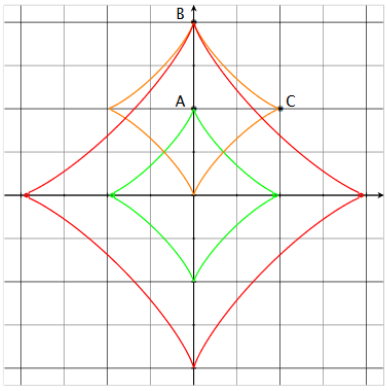
**Distance metric** Given an instance space $\mathcal{X}$, a *distance metric* is a function $\mathrm{Dis} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that for any $x, y, z \in \mathcal{X}$:

- distances between a point and itself are zero: $\mathrm{Dis}(x, x) = 0$;
- all other distances are larger than zero: if $x \neq y$ then $\mathrm{Dis}(x, y) > 0$;
- distances are symmetric: $\mathrm{Dis}(y, x) = \mathrm{Dis}(x, y)$;
- detours can not shorten the distance: $\mathrm{Dis}(x, z) \leq \mathrm{Dis}(x, y) + \mathrm{Dis}(y, z)$.

If the second condition is weakened to a non-strict inequality – i.e., $\mathrm{Dis}(x, y)$ may be zero even if $x \neq y$ – the function $\mathrm{Dis}$ is called a *pseudo-metric*.




The green circle connects points the same Euclidean distance (i.e., Minkowski distance of order $p = 2$) away from the origin as A. The orange circle shows that B and C are equidistant from A. The red circle demonstrates that C is closer to the origin than B, which conforms to the triangle inequality.




With Manhattan distance ($p = 1$), B and C are equally close to the origin and also equidistant from A. With $p < 1$ (here, $p = 0.8$) C is further away from the origin than B; since both are again equidistant from A, it follows that travelling from the origin to C via A is quicker than going there directly, which violates the triangle inequality.



## Means and distances

The arithmetic mean minimises squared Euclidean distance *The arithmetic mean $\boldsymbol{\mu}$ of a set of data points $D$ in a Euclidean space is the unique point that minimises the sum of squared Euclidean distances to those data points.*

Proof. We will show that $\arg\min_{\mathbf{y}} \sum_{\mathbf{x}\in D} ||\mathbf{x} - \mathbf{y}||^2 = \boldsymbol{\mu}$, where $||\cdot||$ denotes the 2-norm. We find this minimum by taking the gradient (the vector of partial derivatives with respect to $y_i$) of the sum and setting it to the zero vector:

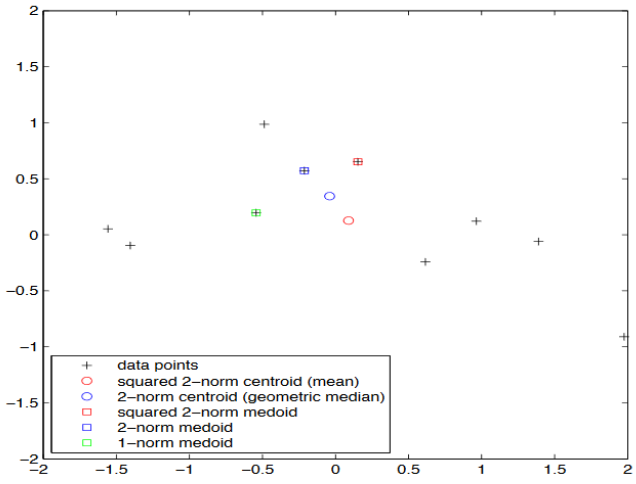
$$\nabla_{\mathbf{y}} \sum_{\mathbf{x}\in D} ||\mathbf{x} - \mathbf{y}||^2 = -2 \sum_{\mathbf{x}\in D} (\mathbf{x} - \mathbf{y}) = -2 \sum_{\mathbf{x}\in D} \mathbf{x} + 2|D|\mathbf{y} = \mathbf{0}$$

from which we derive $\mathbf{y} = \frac{1}{|D|} \sum_{\mathbf{x}\in D} \mathbf{x} = \boldsymbol{\mu}$.

- Notice that minimising the sum of squared Euclidean distances of a given set of points is the same as minimising the average squared Euclidean distance.
- You may wonder what happens if we drop the square here: wouldn't it be more natural to take the point that minimises total Euclidean distance as exemplar?

- An exemplar is a point on the data set which can be used to represent the dataset,
- This point is known as the geometric median, as for univariate data it corresponds to the median or ‘middle value’ of a set of numbers.However, for multivariate data there is no closed-form expression for the geometric median, which needs to be calculated by successive approximation.
- In certain situations it makes sense to restrict an exemplar to be one of the given data points.  In that case, we speak of a medoid, to distinguish it from a centroid which is an exemplar that doesn’t have to occur in the data.
- medoids are always restricted to be members of the data set. Medoids are most commonly used on data when a mean or centroid cannot be defined, such as graphs. They are also used in contexts where the centroid is not representative of the dataset like in images and 3-D trajectories and gene expression
- Finding a medoid requires us to calculate, for each data point, the total distance to all other data points, in order to choose the point that minimises it.  Regardless of the distance metric used, this is an O(n^2) operation fornpoints.
    - This essentially finds the centre of mass of all points, the point where every other point is least distant away from it.
- So for medoids there is no computational reason to prefer one distance metric over another.
- There may be more than one medoid.

## Centroids and mediods

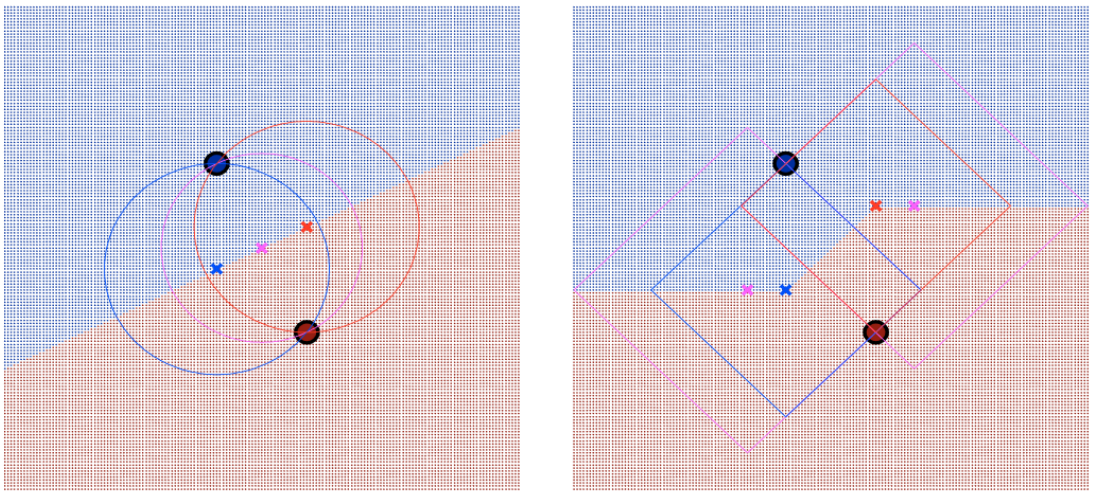



A small data set of 10 points, with circles indicating centroids and squares indicating medoids (the latter must be data points), for different distance metrics. Notice how the outlier on the bottom-right ‘pulls’ the mean away from the geometric median; as a result the corresponding medoid changes as well.

## The basic linear classifier is distance-based

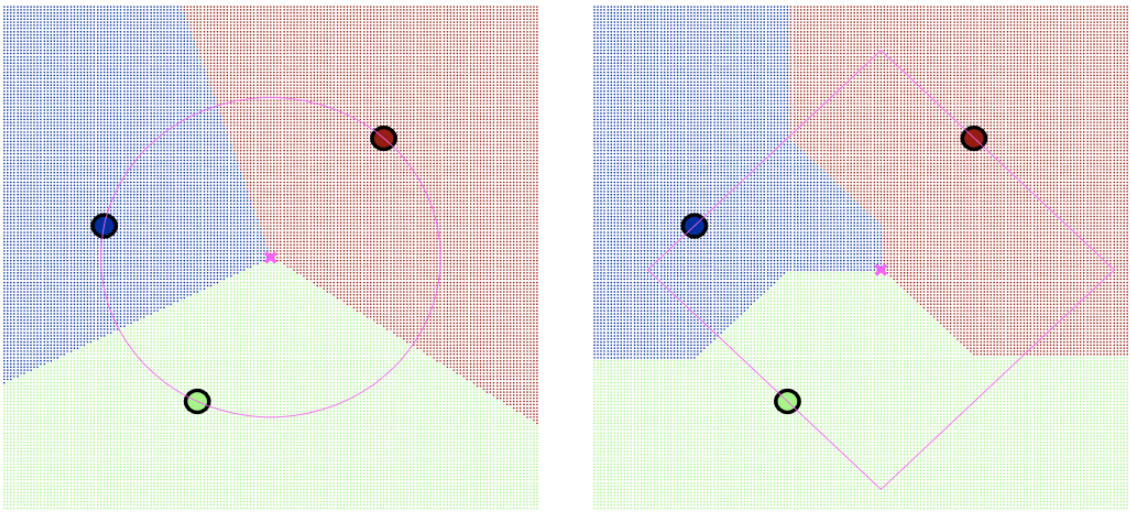
- The basic linear classifier constructs the decision boundary as the perpendicular bisector of the line segment connecting the two exemplars (one for each class).
- An alternative, distance-based way to classify instances without direct reference to a decision boundary is by the following decision rule: if x is nearest to $\mu^{\oplus}$ then classify it as positive, otherwise as negative; or equivalently, classify an instance to the class of the nearest exemplar.
- If we use Euclidean distance as our closeness measure, simple geometry tells us we get exactly the same decision boundary.
- So the basic linear classifier can be interpreted from a distance-based perspective as constructing exemplars that minimise squared Euclidean distance within each class, and then applying a nearest-exemplar decision rule.

## Two-exemplar decision boundaries



(left) For two exemplars the nearest-exemplar decision rule with Euclidean distance results in a linear decision boundary coinciding with the perpendicular bisector of the line connecting the two exemplars. (right) Using Manhattan distance the circles are replaced by diamonds.

## Three-exemplar decision boundaries



(left) Decision regions defined by the 2-norm nearest-exemplar decision rule for three exemplars. (right) With Manhattan distance the decision regions become non-convex.

## Distance-based models

To summarise, the main ingredients of distance-based models are:

- distance metrics, which can be Euclidean, Manhattan, Minkowski orMahalanobis, among many others;
- exemplars: centroids that find a centre of mass according to a chosen distance metric, or medoids that find the most centrally located datapoint; and
- distance-based decision rules, which take a vote among the k nearest exemplars.

## Nearest neighbour classification

- Related to the simplest form of learning: rote learning or memorization
    - Training instances are searched for instance that most closely resembles new or query instance
    - The instances themselves represent the knowledge
    - Called: instance-based, memory-based learning or case-based learning; often a form of local learning
- The similarity or distance function defines “learning”, i.e., how to go beyond simple memorization

- Intuitive idea — instances “close by”, i.e., neighbours or exemplars,should be classified similarly
- Instance-based learning is lazy learning
- Methods: nearest-neighbour, k-nearest-neighbour, lowess, …
- Ideas also important for unsupervised methods, e.g., clustering (later lectures)

Stores all training examples $\langle x_i, f(x_i)\rangle$.

Nearest neighbour:

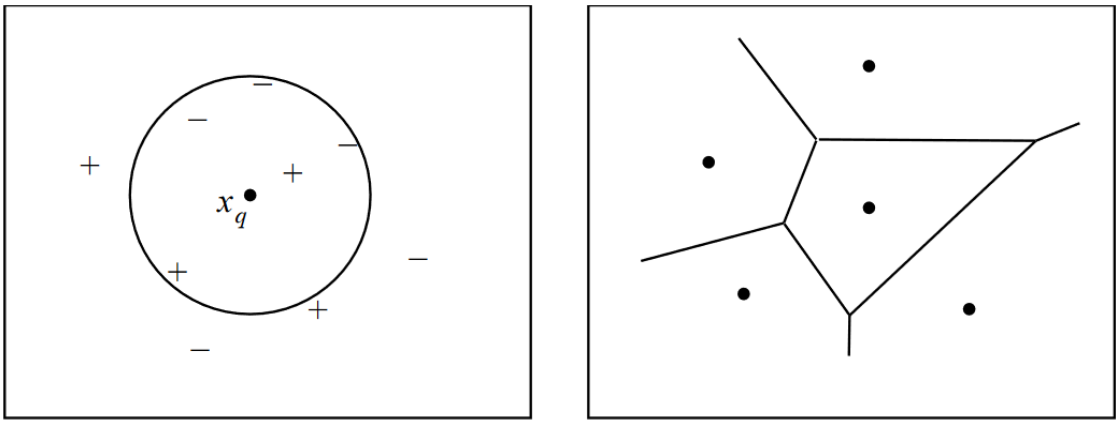
- Given query instance $x_q$, first locate nearest training example $x_n$, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

$k$-Nearest neighbour:

- Given $x_q$, take vote among its $k$ nearest neighbours (if discrete-valued target function)
- take mean of $f$ values of $k$ nearest neighbours (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

## Hypothesis Space for Nearest Neighbour




2 classes, $+$ and $-$ and query point $x_q$. On left, note effect of varying $k$. On right, $1-$NN induces a Voronoi tessellation of the instance space. Formed by the perpendicular bisectors of lines between points.

## Distance Function

The distance function defines what is learned.
Instance $x$ is described by a feature vector (list of attribute-value pairs)

$$\langle a_1(x), a_2(x), \ldots a_m(x) \rangle$$

where $a_r(x)$ denotes the value of the $r$th attribute of $x$.
Most commonly used distance function is *Euclidean* distance ...

- distance between two instances $x_i$ and $x_j$ is defined to be

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{m} (a_r(x_i) - a_r(x_j))^2}$$

Many other distance functions could be used ...

- e.g., *Manhattan* or *city-block* distance (sum of absolute values of differences between attributes)

$$d(x_i, x_j) = \sum_{r=1}^{m} |a_r(x_i) - a_r(x_j)|$$

Vector-based formalization – use *norm* $L_1$, $L_2$, ...
The idea of distance functions will appear again in *kernel methods*.

## Normalization and other issues

- Different attributes measured on different scales
- Need to be *normalized* (why ?)

$$a_r = \frac{v_r - \min v_r}{\max v_r - -\min v_r}$$

  where $v_r$ is the actual value of attribute $r$
- Nominal attributes: distance either 0 or 1
- Common policy for missing values: assumed to be maximally distant (given normalized attributes)

We need to make sure that for non discretized values, that much larger measures e.g. number of sand particles, are not dominating the decision metric vs another feature such as number of children.

## When to consider nearest neighbour

- Instances map to points in R^n
- Less than 20 attributes per instance
  - or number of attributes can be reduced . . .
- Lots of training data
- No requirement for “explanatory” model to be learned

Advantages:

- Statisticians have used k-NN since early 1950s
- Can be very accurate
  - at most twice the “Bayes error” for 1-NN (Cover & Hart, 1967)
- Training is very fast
- Can learn complex target functions
- Don’t lose information by generalization - keep all instances

Disadvantages:

- Slow at query time: basic algorithm scans entire training data to derive a prediction
- Curse of dimensionality
- Assumes all attributes are equally important, so easily fooled by irrelevant attributes
  - Remedy: attribute selection or weights
- Problem of noisy instances:

- Remedy: remove from data set
- not easy – how to know which are noisy ?

## What is the inductive bias of k-NN ?

- an assumption that the classification of query instance x_q will be most similar to the classification of other instances that are nearby according to the distance function

k-NN uses terminology from statistical pattern recognition (see below)

- Regression approximating a real-valued target function
- Residual the error ˆf(x)−f(x) in approximating the target function
- Kernel function of distance used to determine weight of each training example, i.e., kernel function is the function K s.t.
- $$w_i = K(d(x_i, x_q))$$

## Nearest neighbour classifier

- kNN uses the training data as exemplars, so training is O(n) (but prediction is also O(n)!)
- 1NN perfectly separates training data, so low bias but high variance
- By increasing the number of neighbours k we increase bias and decrease variance ( what happens when k=n?)
- Easily adapted to real-valued targets, and even to structured objects(nearest-neighbour retrieval). Can also output probabilities when k >1
- Warning: in high-dimensional spaces everything is far away from everything and so pairwise distances are uninformative (curse of dimensionality)

## Distance-weighted kNN

- Might want to weigh nearer neighbours more heavily …
- Use distance function to construct a weight wi
- Replace the final line of the classification algorithm by:
- https://www.youtube.com/watch?v=xXLTYsHfWzg

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

and $d(x_q, x_i)$ is distance between $x_q$ and $x_i$

-

For real-valued target functions replace the final line of the algorithm by:

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

(denominator normalizes contribution of individual weights).
Now we can consider using *all* the training examples instead of just $k$

$\rightarrow$ using all examples (i.e., when $k = n$) with the rule above is called Shepard's method

-

## Evaluation

Lazy learners do not construct an explicit model, so how do we evaluate the output of the learning process ?

- 1-NN – training set error is always zero !
  - each training example is always closest to itself
- $k$-NN – overfitting may be hard to detect

Leave-one-out cross-validation (LOOCV) – leave out each example and predict it given the rest:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}), (x_{i+1}, y_{i+1}), \ldots, (x_n, y_n)$$

Error is mean over all predicted examples. Fast – no models to be built !

## Curse of dimensionality

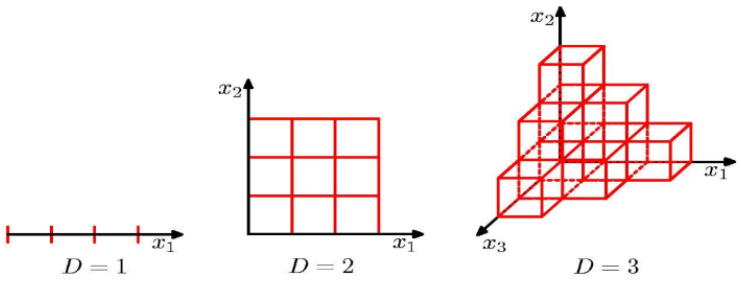
Bellman (1960) coined this term in the context of dynamic programming

Imagine instances described by 20 attributes, but only 2 are relevant to target function — "similar" examples will appear "distant".

*Curse of dimensionality*: nearest neighbour is easily mislead when high-dimensional $X$ – problem of irrelevant attributes

One approach:

- Stretch $j$th axis by weight $z_j$, where $z_1, \ldots, z_n$ chosen to minimize prediction error
- Use cross-validation to automatically choose weights $z_1, \ldots, z_n$
- Note setting $z_j$ to zero eliminates this dimension altogether




- number of "cells" in the instance space grows exponentially in the number of features
- with exponentially many cells we would need exponentially many data points to ensure that each cell is sufficiently populated to make nearest-neighbour predictions reliably

Some ideas to address this for instance-based (nearest-neighbour) learning

- Euclidean distance with weights on attributes

$$\sqrt{\sum w_r(a_r(x_q) - a_r(x))^2}$$

- updating of weights based on nearest neighbour classification error
  - class correct/incorrect: weight increased/decreased
  - can be useful if not all features used in classification

See Moore and Lee (1994) *"Efficient Algorithms for Minimizing Cross Validation Error"*

## Recap

Recap – Practical problems of 1-NN scheme:

- Slow (but fast k−d tree-based approaches exist)
  - Remedy: removing irrelevant data

- Noise (but k-NN copes quite well with noise)
    - Remedy: removing noisy instances
- All attributes deemed equally important
    - Remedy: attribute weighting (or simply selection)
- Doesn't perform explicit generalization
    - Remedy: rule-based or tree-based NN approaches

## Some refinements of instance-based classifiers

- Edited NN classifiers discard some of the training instances before making predictions
- Saves memory and speeds up classification
- IB2: incremental NN learner: only incorporates misclassified instances into the classifier
    - Problem: noisy data gets incorporated
- IB3: store classification performance information with each instance & only use in prediction if above a threshold

## Dealing with noise

Use larger values of k (why ?) How to find the "right" k?

- One way: cross-validation-basedk-NN classifier (but slow)
- Different approach: discarding instances that don't perform well by keeping success records of how well an instance does at prediction (IB3)
    - Computes confidence interval for an instance's success rate and for default accuracy of its class
    - If lower limit of first interval is above upper limit of second one,instance is accepted (IB3: 5%-level)
    - If upper limit of first interval is below lower limit of second one,instance is rejected (IB3: 12.5%-level)

# Classification 2

## Introduction

### Inductive Bias

All models are wrong, but some models are useful.

Confusingly, “inductive bias” is NOT the same “bias” as in the“bias-variance” decomposition.

Inductive bias” is the combination of assumptions and restrictions placed on the models and algorithms used to solve a learning problem.

Essentially it means that the algorithm and model combination you are using to solve the learning problem is appropriate for the task.

Success in machine learning requires understanding the inductive bias of algorithms and models, and choosing them appropriately for the task.

Unfortunately, for most machine learning algorithms it is not always easy to know what their inductive bias is.

For example, what is the inductive bias of:

- Linear Regression
    - Target function has the form y = ax + b
    - Approximate by using MSE
- Nearest Neighbour
    - Target function is a complex non-linear function of the data
    - Predict using nearest neighbour by Euclidean distance in feature space

What we would really like:

- a framework for machine learning algorithms
- With a way of representing the inductive bias
- ideally , should be a declarative specification
- Also should quantify uncertainty in the inductive bias

# A probabilistic approach

## Decision rule

A probabilistic approach Probabilistic models Decision rule Assuming that X and Y are the only variables we know and care about,the posterior distribution P(Y|X) helps us to answer many questions of interest.

- For instance, to classify a new email we determine whether the words ‘Viagra’ and ‘lottery’ occur in it, look up the corresponding probability P(Y=spam|Viagra,lottery), and predict spam if this probability exceeds 0.5 and ham otherwise.
- Such a recipe to predict a value of Y on the basis of the values of X and the posterior distribution P(Y|X) is called a decision rule.

# Bayesian Machine Learning

## Two Roles for Bayesian Methods

Provides practical learning algorithms:

- Naive Bayes classifier learning
- Bayesian network learning, etc.
- Combines prior knowledge (prior probabilities) with observed data
- How to get prior probabilities ?

Provides useful conceptual framework:

- Provides a “gold standard” for evaluating other learning algorithms
- Gives some additional insight into “Occam’s razor

## Basic Formulas for Probabilities

*Product Rule*: probability $P(A \wedge B)$ of conjunction of two events A and B:

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

*Sum Rule*: probability of disjunction of two events A and B:

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

*Theorem of total probability*: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$, then:

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

Also worth remembering:

- *Conditional Probability*: probability of $A$ given $B$:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

- Rearrange sum rule to get:

$$P(A \wedge B) = P(A) + P(B) - P(A \vee B)$$

## Bayes Theorem

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

where

$P(h)$ = prior probability of hypothesis $h$
$P(D)$ = prior probability of training data $D$
$P(h|D)$ = probability of $h$ given $D$
$P(D|h)$ = probability of $D$ given $h$

## Choosing Hypotheses

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Generally, we want the most probable hypothesis given the training data

*Maximum a posteriori* hypothesis $h_{MAP}$:

$$\begin{aligned} h_{MAP} &= \arg\max_{h \in H} P(h|D) \\ &= \arg\max_{h \in H} \frac{P(D|h)P(h)}{P(D)} \\ &= \arg\max_{h \in H} P(D|h)P(h) \end{aligned}$$

If assume $P(h_i) = P(h_j)$ then can further simplify, and choose the *Maximum likelihood* (ML) hypothesis

$$h_{ML} = \arg\max_{h_i \in H} P(D|h_i)$$

## Applying Bayes Theorem

Does patient have cancer or not?

*A patient takes a lab test and the result comes back positive. The test returns a correct positive result in only $98\%$ of the cases in which the disease is actually present, and a correct negative result in only $97\%$ of the cases in which the disease is not present. Furthermore, $.008$ of the entire population have this cancer.*

$$P(cancer) = .008 \qquad P(\neg cancer) = .992$$
$$P(\oplus \mid cancer) = .98 \qquad P(\ominus \mid cancer) = .02$$
$$P(\oplus \mid \neg cancer) = .03 \qquad P(\ominus \mid \neg cancer) = .97$$

Does patient have cancer or not?

We can find the maximum a posteriori (MAP) hypothesis

$$\begin{aligned} P(\oplus \mid cancer)P(cancer) &= 0.98 \times 0.008 = 0.00784 \\ P(\oplus \mid \neg cancer)P(\neg cancer) &= 0.03 \times 0.992 = 0.02976 \end{aligned}$$

Thus $h_{MAP} = \neg cancer$.

Also note: posterior probability of hypothesis $cancer$ higher than prior.

How to get the posterior probability of a hypothesis $h$ ?
Divide by $P(\oplus)$, probability of data, to normalize result for $h$:

$$P(h|D) = \frac{P(D|h)P(h)}{\sum_{h_i \in H} P(D|h_i)P(h_i)}$$

Denominator ensures we obtain posterior probabilities that sum to 1.
Sum for all possible numerator values, since hypotheses are mutually exclusive (e.g., patient either has cancer or does not).
Marginal likelihood (marginalizing out likelihood over all possible values in the hypothesis space) — prior probability of the data.

## A bayesian framework for Classification

First, define a prior on the hypothesis

Next, define define the likelihood, i.e., the probability of the data given the hypothesis

Then learning is finding the required parameters by fitting hypotheses to data

Predict (classify) using, e.g., the MAP hypothesis

## Brute Force MAP Hypothesis Learner

Idea: view learning as finding the *most probable* hypothesis

- For each hypothesis $h$ in $H$, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- Output the hypothesis $h_{MAP}$ with the highest posterior probability

$$h_{MAP} = \arg\max_{h \in H} P(h|D)$$

# A Bayesian approach to learning algorithms

## Relation to Concept Learning (i.e., classification)

Canonical concept learning task:

- instance space X, hypothesis space H, training examples D
- consider a learning algorithm that outputs most specific hypothesis from the version space V S_H,D (i.e., set of all consistent or"zero-error" classification rules)

What would Bayes rule produce as the MAP hypothesis?

Does this algorithm output a MAP hypothesis?

Brute Force MAP Framework for Concept Learning:

Assume fixed set of instances $\langle x_1, \ldots, x_m \rangle$

Assume $D$ is the set of classifications $D = \langle c(x_1), \ldots, c(x_m) \rangle$

Choose $P(h)$ to be *uniform* distribution:

- $P(h) = \frac{1}{|H|}$ for all $h$ in $H$

Choose $P(D|h)$:

- $P(D|h) = 1$ if $h$ consistent with $D$
- $P(D|h) = 0$ otherwise

Then:

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases}$$

Note that since likelihood is zero if $h$ is inconsistent then the posterior is also zero. But how did we obtain the posterior for consistent $h$ ?

$$\begin{aligned} P(h|D) &= \frac{1 \cdot \frac{1}{|H|}}{P(D)} \\ &= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} \\ &= \frac{1}{|VS_{H,D}|} \end{aligned}$$

How did we obtain $P(D)$ ? From theorem of total probability:

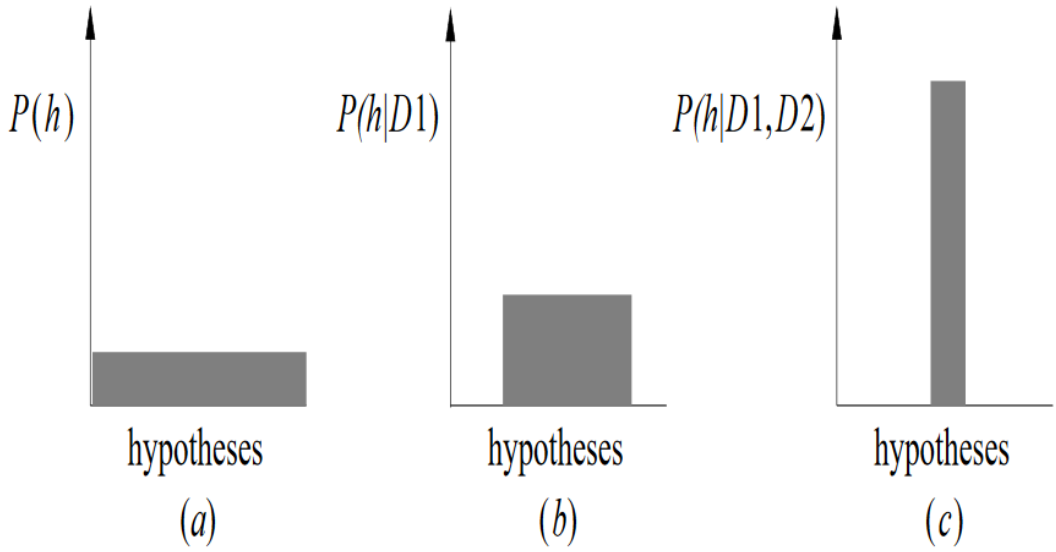
$$
\begin{aligned}
P(D) &= \sum_{h_i \in H} P(D|H_i)P(h_i) \\
&= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|} \\
&= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} \\
&= \frac{|VS_{H,D}|}{|H|}
\end{aligned}
$$

Every hypothesis consistent with D is a MAP hypothesis, if we assume

- uniform probability over H
- target function $c \in H$
- deterministic, noise-free data
- etc. (see above)

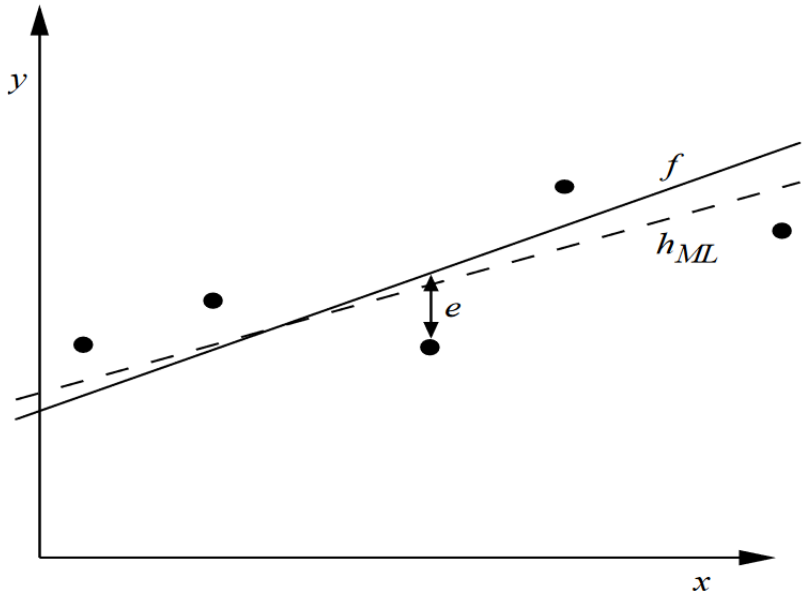
So, this learning algorithm will output a MAP hypothesis, even though it does not explicitly use probabilities in learning.

Evolution of Posterior Probabilities



## Learning A real Valued Function

E.g., learning a linear target function $f$ from noisy examples:




Consider any real-valued target function $f$
Training examples $\langle x_i, d_i \rangle$, where $d_i$ is noisy training value

- $d_i = f(x_i) + e_i$
- $e_i$ is random variable (noise) drawn independently for each $x_i$ according to some Gaussian (normal) distribution with mean=0

Then the **maximum likelihood** hypothesis $h_{ML}$ is the one that **minimizes the sum of squared errors**:

$$h_{ML} = \arg\min_{h \in H} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

How did we obtain this ?

$$\begin{aligned} h_{ML} &= \arg\max_{h \in H} p(D|h) \\ &= \arg\max_{h \in H} \prod_{i=1}^{m} p(d_i|h) \\ &= \arg\max_{h \in H} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\frac{d_i - h(x_i)}{\sigma})^2} \end{aligned}$$

Recall that we treat each probability $p(D|h)$ **as if** $h = f$, i.e., **we assume** $\mu = f(x_i) = h(x_i)$, which is the key idea behind maximum likelihood !

Maximize natural log to give simpler expression:

$$\begin{aligned} h_{ML} &= \arg\max_{h \in H} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2}\left(\frac{d_i - h(x_i)}{\sigma}\right)^2 \\ &= \arg\max_{h \in H} \sum_{i=1}^{m} -\frac{1}{2}\left(\frac{d_i - h(x_i)}{\sigma}\right)^2 \\ &= \arg\max_{h \in H} \sum_{i=1}^{m} -\left(d_i - h(x_i)\right)^2 \end{aligned}$$

Equivalently, we can minimize the positive version of the expression:

$$h_{ML} = \arg\min_{h \in H} \sum_{i=1}^{m} \left(d_i - h(x_i)\right)^2$$

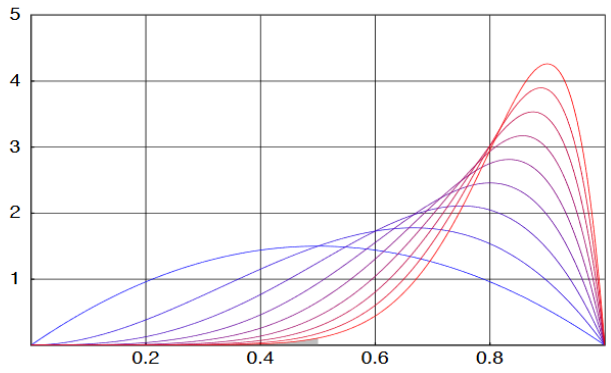
## Discriminative and generative probabilistic models

- *Discriminative models* model the posterior probability distribution $P(Y|X)$, where $Y$ is the target variable and $X$ are the features. That is, given $X$ they return a probability distribution over $Y$.
- *Generative models* model the joint distribution $P(Y, X)$ of the target $Y$ and the feature vector $X$. Once we have access to this joint distribution we can derive any conditional or marginal distribution involving the same variables. In particular, since $P(X) = \sum_y P(Y = y, X)$ it follows that the posterior distribution can be obtained as $P(Y|X) = \frac{P(Y,X)}{\sum_y P(Y=y,X)}$.
- Alternatively, generative models can be described by the likelihood function $P(X|Y)$, since $P(Y, X) = P(X|Y)P(Y)$ and the target or prior distribution (usually abbreviated to 'prior') can be easily estimated or postulated.
- Such models are called 'generative' because we can sample from the joint distribution to obtain new data points together with their labels. Alternatively, we can use $P(Y)$ to sample a class and $P(X|Y)$ to sample an instance for that class.

## Assessing uncertainty in estimates

Suppose we want to estimate the probability $\theta$ that an arbitrary e-mail is spam, so that we can use the appropriate prior distribution.

- The natural thing to do is to inspect $n$ e-mails, determine the number of spam e-mails $d$, and set $\hat{\theta} = d/n$; we don't really need any complicated statistics to tell us that.
- However, while this is the most likely estimate of $\theta$ – the maximum a posteriori (MAP) estimate – this doesn't mean that other values of $\theta$ are completely ruled out.
- We model this by a probability distribution over $\theta$ (a Beta distribution in this case) which is updated each time new information comes in. This is further illustrated in the figure for a distribution that is more and more skewed towards spam.
- For each curve, its bias towards spam is given by the area under the curve and to the right of $\theta = 1/2$.




Each time we inspect an e-mail, we are reducing our uncertainty regarding the prior spam probability $\theta$. After we inspect two e-mails and observe one spam, the possible $\theta$ values are characterised by a symmetric distribution around $1/2$. If we inspect a third, fourth, ..., tenth e-mail and each time (except the first one) it is spam, then this distribution narrows and shifts a little bit to the right each time. The distribution for $n$ e-mails reaches its maximum at $\hat{\theta}_{\mathrm{MAP}} = \frac{n-1}{n}$ (e.g., $\hat{\theta}_{\mathrm{MAP}} = 0.8$ for $n = 5$).

## The Bayesian perspective

Explicitly modelling the posterior distribution over the parameter θ has a number of advantages that are usually associated with the 'Bayesian' perspective:

- We can precisely characterise the uncertainty that remains about our estimate by quantifying the spread of the posterior distribution.
- We can obtain a generative model for the parameter by sampling from the posterior distribution, which contains much more information than a summary statistic such as the MAP estimate canconvey – so, rather than using a single e-mail with θ=θMAP, our generative model can contain a number of e-mails with θ sampled from the posterior distribution.
- We can quantify the probability of statements such as 'e-mails are biased towards ham' (the tiny shaded area in the figure demonstrates that after observing one ham and nine spam e-mails this probability is very small, about 0.6%).
- We can use one of these distributions to encode our prior beliefs: e.g.,if we believe that the proportions of spam and ham are typically 50–50, we can take the distribution for n= 2 (the lowest, symmetric one in the figure on the previous slide) as our prior.

The key point is that probabilities do not have to be interpreted as estimates of relative frequencies, but can carry the more general meaning of (possibly subjective) degrees of belief. Consequently, we can attach a probability distribution to almost anything: not just features and targets, but also model parameters and even models.

## Minimum Description Length Principle

Once again, the MAP hypothesis

$$h_{MAP} = \arg\max_{h \in H} P(D|h)P(h)$$

Which is equivalent to

$$h_{MAP} = \arg\max_{h \in H} \log_2 P(D|h) + \log_2 P(h)$$

Or

$$h_{MAP} = \arg\min_{h \in H} -\log_2 P(D|h) - \log_2 P(h)$$

Interestingly, this is an expression about a quantity of *bits*.

$$h_{MAP} = \arg\min_{h \in H} -\log_2 P(D|h) - \log_2 P(h) \quad (1)$$

From information theory:

> *The optimal (shortest expected coding length) code for an event with probability $p$ is $-\log_2 p$ bits.*

So interpret (1):

- $-\log_2 P(h)$ is length of $h$ under optimal code
- $-\log_2 P(D|h)$ is length of $D$ given $h$ under optimal code

**Note well:** assumes *optimal* encodings, when the priors and likelihoods are known. In practice, this is difficult, and makes a difference.

# Performance of Bayes classifiers

## Occam's Razor

We always prefer the shortest hypothesis (avoids overfitting)

- Prefer the hypothesis h that minimizes

$$h_{MDL} = \arg\min_{h \in H} L_{C_1}(h) + L_{C_2}(D|h)$$

Where Lc(x) is the description length of x under the optimal encoding C

## Most Probable Classification

Given new instances of x, hMAP(x) is not the most probable classification. Note hMAP(x) only gives us the most probable outcome given the data, not the most probable classification in a general sense. Consider the following example (we can pretend h are class outputs)

- Three hypothesis

$$P(h_1|D) = .4, \; P(h_2|D) = .3, \; P(h_3|D) = .3$$

- Given new instance of x

$$h_1(x) = +, \; h_2(x) = -, \; h_3(x) = -$$

How do we classify x? To do this we use Bayes optimal classification formulae

$$\arg\max_{v_j \in V} \sum_{h_i \in H} P(v_j|h_i) P(h_i|D)$$

The formula goes through each possible class output in the space V, and with all the hypothesis H it will go through each hypothesis, and check the posterior probability of our hypothesis given the data multiplied by the probability of the class output given the hypothesis, and pick the highest.

Applying this to our example we have the following values

$$P(h_1|D) = .4, \quad P(-|h_1) = 0, \quad P(+|h_1) = 1$$
$$P(h_2|D) = .3, \quad P(-|h_2) = 1, \quad P(+|h_2) = 0$$
$$P(h_3|D) = .3, \quad P(-|h_3) = 1, \quad P(+|h_3) = 0$$

Plugging these values into the algorithm gives us

$$\sum_{h_i \in H} P(+|h_i)P(h_i|D) = .4$$

$$\sum_{h_i \in H} P(-|h_i)P(h_i|D) = .6$$

$$\arg\max_{v_j \in V} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) = -$$

Despite hypothesis 1 having the highest hMAP(x) value, it gave an incorrect classification, and the - output is the more probable classification.
There is no other classification method using the same hypothesis space and same prior knowledge that can outperform this method on average.

## Bayes Error

The probability of error for classifying an instance x for a two class classifier is the following

$$P(\text{error}|x) = P(\text{class}_1|x) \quad \text{if we predict class}_2$$
$$= P(\text{class}_2|x) \quad \text{if we predict class}_1$$

Given a set of output classes x, the probability of error can be calculated simply by summing all the possible errors given the following.

$$\Sigma_x \quad P(\text{error}) = P(\text{error}|x)\ P(x)$$

This allows us to justify the use of the decision rule, that if the probability of class 1 is larger than the probability of class 2, we classify as class 1. This is because we are picking the class which minimises the probability of classification error. We are also able to give confidence levels on our prediction.

## Naive Bayes

We tend to use Naive Bayes for

- Moderate or large training sets
- Attributes that describes instances that are conditionally independent given classification
    - We want feature independence, no correlation between features which affect outcome

Places were Naive Bayes is used and is extremely successful is

- Classifying text document
- Word prediction
- Guassian Naive Bayes for real-values data.

In a simple sense all we do is take the class with the highest probability output given the features supplied, same as hMAP

$$v_{MAP} = \arg\max_{v_j \in V} P(v_j|a_1, a_2 \ldots a_n)$$

And we can simply reduce this to

$$\begin{aligned} v_{MAP} &= \arg\max_{v_j \in V} \frac{P(a_1, a_2 \ldots a_n|v_j)P(v_j)}{P(a_1, a_2 \ldots a_n)} \\ &= \arg\max_{v_j \in V} P(a_1, a_2 \ldots a_n|v_j)P(v_j) \end{aligned}$$

## Naive Bayes Assumption

Naive Bayes works under the assumption that

- Attributes are statistically independent given class value
    - This means that knowing the value of a particular attribute tells us nothing about the value of another attribute if the class is known

This can be seen as each feature having independent probability

$$P(a_1, a_2 \dots a_n | v_j) = \prod_i P(a_i | v_j)$$

This gives us the classifier working in the following way, we pick the class that gives us the highest likelihood when checking the probability of all features in respect to that class

$$\textbf{Naive Bayes classifier: } v_{NB} = \arg\max_{v_j \in V} P(v_j) \prod_i P(a_i | v_j)$$

## Bernoulli Naive Bayes

Suppose we had a coin toss simulation where X represents the outcome of n coin tosses. If the probability of heads is given by θ, then X has a binomial distribution of the following (2 outcomes)

$$\text{Bin}(k|n, \theta) = \binom{n}{k} \theta^k (1-\theta)^{n-k}$$

For a single coin toss, X has the value either 0 or value 1, if the probability of heads is θ, then X has a bernoulli distribution of the following

$$\text{Ber}(x|\theta) = \theta^{\mathbb{I}(x=1)} (1-\theta)^{\mathbb{I}(x=0)}$$

Note I is the indicator, function or flag, if true == 1 else 0

However in data sometimes we get more complicated situations with more features

## Multinomial Naive Bayes

Suppose we now have something that is modelled similar to a k-sided dice throw. Let X be a vector that shows the number of times we land on a face of our k-sided die

$$\text{Let } \mathbf{x} = (x_1, \ldots, x_K)$$

If $\Theta\hat{}_j$ is the probability of side j, then X has a multinomial distribution of

$$\text{Mu}(\mathbf{x}|n, \theta) = \binom{n}{x_1 \ldots x_K} \prod_{j=1}^{K} \theta_j^{x_j}$$

Here x_j is the number of occurrences of index j

For a single dice roll, we will have X as a vector of blank indices, and 1 index will be set to 1, which is the face that was rolled. If the probability of j is as above then X has a multivariate Bernoulli distribution of the following

$$\text{Mu}(\mathbf{x}|1, \theta) = \prod_{j=1}^{K} \theta_j^{\mathbb{I}(x_j=1)}$$

## Categorical features

Categorical features are discrete features which have a finite and set amount of outputs predefined e.g. true/false, similar to enums in C.

- the common form of bernoulli distribution models whether or not a word occurs in a document. So for the i-th word in our vocabulary we have a random variable X_i which is governed by a Bernoulli distribution. The joint distribution over the bit vector

  $$X = (X_1, \ldots, X_k)$$

  Is the multivariate Bernoulli distribution.
- In some cases however we have more than 2 outcomes not just whether a word occurs or not, for example in email, we can have a vector where each index is a word, and the corresponding value is the number of times the word appears, this allows us to make a

histogram showing frequency counts and allow us to influence the classification of a document.

Both these approaches are used commonly and they assume independence between word occurrences (Naive Bayes)

- In the case of the multinomial document model, we assume words at different words positions are drawn independently, following from the use of multinomial distribution
- In the case of a multivariate Bernoulli model we assume that the bits in a bit vector are statistically independent. This allows us to predict the joint probability of a particular bit being on as the product of the probabilities of each component

$$P(X_i = x_i).$$

- In practice these assumptions often aren't true, there is often correlation between words e.g. if we see the word "viagra" we can almost be certain the word "pill" will be contained. Violated assumptions of independence will reduce the quality of probability estimate however, we still possibly get good performance.

## Decision Rules

We have chosen one of the distribution models for our data X

- The more different $P(X|Y = \text{spam})$ and $P(X|Y = \text{ham})$ are, the more useful the features $X$ are for classification.
- Thus, for a specific e-mail $x$ we calculate both $P(X = x|Y = \text{spam})$ and $P(X = x|Y = \text{ham})$, and apply one of several possible decision rules:

We can choose from the following decision rules

- Maximum likelihood (ML) - predict the class which has the highest likelihood to match those features

$$\text{predict } \arg\max_y P(X = x|Y = y);$$

- Maximum a posteriori (MAP) - predict same as maximum likelihood, but take into account probability of class

$$\arg\max_y P(X = x|Y = y)P(Y = y);$$

Under the assumption that there is uniform class distribution, ML and MAP have the exact same prediction.

## Example using NB

### Multivariate Bernoulli model

Suppose we have a vocabulary of only three words a, b and c. Suppose we use a multivariate Bernoulli model for our spam/ham email filter with the following parameters. (+ = spam, - = ham)

$$\boldsymbol{\theta}^{\oplus} = (0.5, 0.67, 0.33) \qquad \boldsymbol{\theta}^{\ominus} = (0.67, 0.33, 0.33)$$

We can just see already that the presence of the word b is twice as common in spam vs ham. Suppose we have an email that needs to be classified but it only contains words a and b. So we have the following vector (1, 1, 0). We obtain the likelihoods in the following

$$P(\mathbf{x}|\oplus) = 0.5 \cdot 0.67 \cdot (1 - 0.33) = 0.222$$
$$P(\mathbf{x}|\ominus) = 0.67 \cdot 0.33 \cdot (1 - 0.33) = 0.148$$

Note since c does not occur, we take 1 - probability(c)

Using this we can make the decision that the email is spam.

### Multinomial model

Same as above however with these parameters

$$\boldsymbol{\theta}^{\oplus} = (0.3, 0.5, 0.2) \qquad \boldsymbol{\theta}^{\ominus} = (0.6, 0.2, 0.2)$$

The email we are classifying has 3 occurrences of the word a, one occurrence of the word b and no occurrences of the word c, giving us the vector (3, 1, 0). The total number of vocabulary word occurrences is given by n = 4 (sum of vector). Using this we calculate the likelihoods

$$P(\mathbf{x}|\oplus) = 4! \frac{0.3^3}{3!} \frac{0.5^1}{1!} \frac{0.2^0}{0!} = 0.054$$
$$P(\mathbf{x}|\ominus) = 4! \frac{0.6^3}{3!} \frac{0.2^1}{1!} \frac{0.2^0}{0!} = 0.1728$$

Using this decision rule we can see x is classified as ham, opposite of multivariate bernoulli model. This is due to the fact that the frequency of the word a provides strong evidence of ham.

The denominators are the occurrences factorial.
The likelihood ratio given our sample and our parameter is the following which does not swing our decision that it is spam since it is < 1, the likelihood is much more in favour of the hamclass than in the spam class.

$$\left(\frac{0.3}{0.6}\right)^3 \left(\frac{0.5}{0.2}\right)^1 \left(\frac{0.2}{0.2}\right)^0 = 5/16.$$

## Likelihood ratio

In the case of two classes, it is convenient to work with likelihoods and odds. For the example above the likelihood ratio can be calculated with the following

$$\frac{P(\mathbf{x}|\oplus)}{P(\mathbf{x}|\ominus)} = \frac{0.5}{0.67}\frac{0.67}{0.33}\frac{1-0.33}{1-0.33} = 3/2 > 1$$

This means that the MAP classification of x is also spam if the prior odds are more than ⅔ (since when we multiple with prior will be >= 1), but ham if they are less than ⅔

- For example with 33% spam and 67% ham, prior odds are the following

$\frac{P(\oplus)}{P(\ominus)} = \frac{0.33}{0.67} = 1/2$, resulting in a posterior odds of

$$\frac{P(\oplus|\mathbf{x})}{P(\ominus|\mathbf{x})} = \frac{P(\mathbf{x}|\oplus)}{P(\mathbf{x}|\ominus)}\frac{P(\oplus)}{P(\ominus)} = 3/2 \cdot 1/2 = 3/4 < 1$$

Since the likelihood ratio for x is not strong enough to push the decision away from the prior we stick to our original classification. If our ratio however was > 1 it would swing the decision for our classifier.

- For instance, suppose that for a particular e-mail described by $X$ we have $P(X|Y = \text{spam}) = 3.5 \cdot 10^{-5}$ and $P(X|Y = \text{ham}) = 7.4 \cdot 10^{-6}$, then observing $X$ in a spam e-mail is nearly five times more likely than it is in a ham e-mail.

The likelihood ratio suggests a decision rule that predicts spam if likelihood ratio > 1, or ham otherwise.

It is worth noting that this multivariate likelihood ratio is different to the multinomial likelihood ratio of 5/16.

It is worth noting that despite multinomial using frequencies and making more use out of the data, it may seem like the always go to method, however we must take into account this adds weighting to the frequencies, and in cases it may not be good, and if the frequencies is not so important, we would end up with a worse model. Choosing between the two is made outside the learning algorithm and depends on the data and task.

### When to use likelihoods

Likelihoods allow us to ignore the prior distributions or assume it uniform, this is particularly useful if we have a lot of data, where the prior is not so important. However if we don't have a lot of data using the prior might be useful.

## Training for NB

Depending whether we use multinomial or multivariate models, our fitting and parameterisation will work differently.

- Multinomial

| E-mail | $\#a$ | $\#b$ | $\#c$ | Class |
|---|---|---|---|---|
| $e_1$ | 0 | 3 | 0 | $+$ |
| $e_2$ | 0 | 3 | 3 | $+$ |
| $e_3$ | 3 | 0 | 0 | $+$ |
| $e_4$ | 2 | 3 | 0 | $+$ |
| $e_5$ | 4 | 3 | 0 | $-$ |
| $e_6$ | 4 | 0 | 3 | $-$ |
| $e_7$ | 3 | 0 | 0 | $-$ |
| $e_8$ | 0 | 0 | 0 | $-$ |

- Sum up the count vectors for each class (5, 9, 3) for spam, (11, 3, 3) for ham
- To avoid a major problems of no occurrences which zeros out our probabilities we must add a pseudo-count of one to each count to make sure that there are no zero occurrences
- Using the count vectors and the total count 20 (including pseudo-count) we get the following parameters

  - The estimated parameter vectors are thus
    $\hat{\boldsymbol{\theta}}^{\oplus} = (6/20, 10/20, 4/20) = (0.3, 0.5, 0.2)$ for spam and
    $\hat{\boldsymbol{\theta}}^{\ominus} = (12/20, 4/20, 4/20) = (0.6, 0.2, 0.2)$ for ham.

- Multivariate bernoulli model

| E-mail | $a?$ | $b?$ | $c?$ | Class |
|---|---|---|---|---|
| $e_1$ | 0 | 1 | 0 | $+$ |
| $e_2$ | 0 | 1 | 1 | $+$ |
| $e_3$ | 1 | 0 | 0 | $+$ |
| $e_4$ | 1 | 1 | 0 | $+$ |
| $e_5$ | 1 | 1 | 0 | $-$ |
| $e_6$ | 1 | 0 | 1 | $-$ |
| $e_7$ | 1 | 0 | 0 | $-$ |
| $e_8$ | 0 | 0 | 0 | $-$ |

  - Here emails are represented by bit vectors
    - Adding the bit vectors for each class results in (2, 3, 1) for spam and (3, 1, 1) for ham
    - Probability smoothing now means adding two pseudo-documents, one containing each word and one containing now, vector (1, 1, 1, 1) and (0, 0 ,0 ,0).
    - Each count is divided by the number of samples in a class (since we added 2 more examples we went from 4 to 6
    - This gives us our parameters as the following
      - This results in the estimated parameter vectors $\hat{\boldsymbol{\theta}}^{\oplus} = (3/6, 4/6, 2/6) = (0.5, 0.67, 0.33)$ for spam and $\hat{\boldsymbol{\theta}}^{\ominus} = (4/6, 2/6, 2/6) = (0.67, 0.33, 0.33)$ for ham.

## The Algorithm

Naive_Bayes_Learn($examples$)

For each target value $v_j$

$\hat{P}(v_j) \leftarrow$ estimate $P(v_j)$

For each attribute value $a_i$ of each attribute $a$

$\hat{P}(a_i|v_j) \leftarrow$ estimate $P(a_i|v_j)$

Classify_New_Instance($x$)

$$v_{NB} = \arg\max_{v_j \in V} \hat{P}(v_j) \prod_{a_i \in x} \hat{P}(a_i|v_j)$$

## Posterior Odds

$$\frac{P(Y = \text{spam}|\text{Viagra} = 0, \text{lottery} = 0)}{P(Y = \text{ham}|\text{Viagra} = 0, \text{lottery} = 0)} = \frac{0.31}{0.69} = 0.45$$

$$\frac{P(Y = \text{spam}|\text{Viagra} = 1, \text{lottery} = 1)}{P(Y = \text{ham}|\text{Viagra} = 1, \text{lottery} = 1)} = \frac{0.40}{0.60} = 0.67$$

$$\frac{P(Y = \text{spam}|\text{Viagra} = 0, \text{lottery} = 1)}{P(Y = \text{ham}|\text{Viagra} = 0, \text{lottery} = 1)} = \frac{0.65}{0.35} = 1.9$$

$$\frac{P(Y = \text{spam}|\text{Viagra} = 1, \text{lottery} = 0)}{P(Y = \text{ham}|\text{Viagra} = 1, \text{lottery} = 0)} = \frac{0.80}{0.20} = 4.0$$

Using a MAP decision rule we predict ham in the top two cases and spam in the bottom two. Given that the full posterior distribution is all there is to know about the domain in a statistical sense, these predictions are the best we can do: they are *Bayes-optimal*.

Note < 1 = negative case, >1 is the positive case.

When applying the naive bayes method of taking independent probabilities and finding marginal likelihoods we get the following

$$\frac{P(\text{Viagra} = 0|Y = \text{spam})}{P(\text{Viagra} = 0|Y = \text{ham})} \frac{P(\text{lottery} = 0|Y = \text{spam})}{P(\text{lottery} = 0|Y = \text{ham})} = \frac{0.60}{0.88} \frac{0.79}{0.87} = 0.62 \quad (0.45)$$

$$\frac{P(\text{Viagra} = 0|Y = \text{spam})}{P(\text{Viagra} = 0|Y = \text{ham})} \frac{P(\text{lottery} = 1|Y = \text{spam})}{P(\text{lottery} = 1|Y = \text{ham})} = \frac{0.60}{0.88} \frac{0.21}{0.13} = 1.1 \quad (1.9)$$

$$\frac{P(\text{Viagra} = 1|Y = \text{spam})}{P(\text{Viagra} = 1|Y = \text{ham})} \frac{P(\text{lottery} = 0|Y = \text{spam})}{P(\text{lottery} = 0|Y = \text{ham})} = \frac{0.40}{0.12} \frac{0.79}{0.87} = 3.0 \quad (4.0)$$

$$\frac{P(\text{Viagra} = 1|Y = \text{spam})}{P(\text{Viagra} = 1|Y = \text{ham})} \frac{P(\text{lottery} = 1|Y = \text{spam})}{P(\text{lottery} = 1|Y = \text{ham})} = \frac{0.40}{0.12} \frac{0.21}{0.13} = 5.4 \quad (0.67)$$

Here the same classification is given in the first 3 cases, however in the last case it is highly favoured towards spam, opposite of the bayes optimal classifier. This illustrates the issue with naive assumptions that assumes all features are independent.

Conditional independence assumption is often not true, however it still works surprisingly well, we don't need our probability of our posteriors to be correct, we only want the classification to be relatively the same.

$$\arg\max_{v_j \in V} \hat{P}(v_j) \prod_i \hat{P}(a_i|v_j) = \arg\max_{v_j \in V} P(v_j) P(a_1 \ldots, a_n|v_j)$$

This says that as long as we take the maximum from choosing independent probabilities, and it matches the maximum from the probability of the dependent probabilities, we still get the same classification.

Having too many redundant features will cause problems (e.g. identical attributes) and this will skew our probability of our posterior. It is important to perform feature selection or data processing to make sure we aren't working with redundant features.

## Numeric Attributes for NB

When working with numeric attributes we work under the assumption that attributes have a normal (Gaussian) probability distribution given the class (bell curve).

The probability density function is defined by the mean and standard definition in the following

The sample mean $\mu$:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The standard deviation $\sigma$:

$$\sigma = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2$$

Now we have the following function to work out probability density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Example: continuous attribute *temperature* with mean = 73 and standard deviation = 6.2. Density value

$$f(\text{temperature} = 66 | \text{“yes”}) = \frac{1}{\sqrt{2\pi}6.2} e^{-\frac{(66-73)^2}{2\times 6.2^2}} = 0.0340$$

Missing values during training are not included in calculation of mean and standard deviation.

Note that these values for mean and standard evaluation are calculated on the class yes with the temperature attribute, we would need to do the same for no and all other attributes.

## Classifying Documents with NB

In this example we will

- Learn which news articles are interesting
- Learn to classify web pages by topic

This is the most common application for Naive Bayes and it is one of the most effective algorithms for the task.

For representing text documents we will use the following attributes.

- Vector of words x where each attribute is a unique word in the document (can include frequency of word if needed)

To use Naive Bayes we first need to calculate the following probabilities

- $P(+)$
- $P(-)$
- $P(doc|+)$
- $P(doc|-)$

Where doc is just our vector of words x.
We make the following Naive Bayes assumption that all words are independent attributes

$$P(doc|v_j) = \prod_{i=1}^{length(doc)} P(a_i = w_k|v_j)$$

where $P(a_i = w_k|v_j)$ is probability that word in position $i$ is $w_k$, given $v_j$

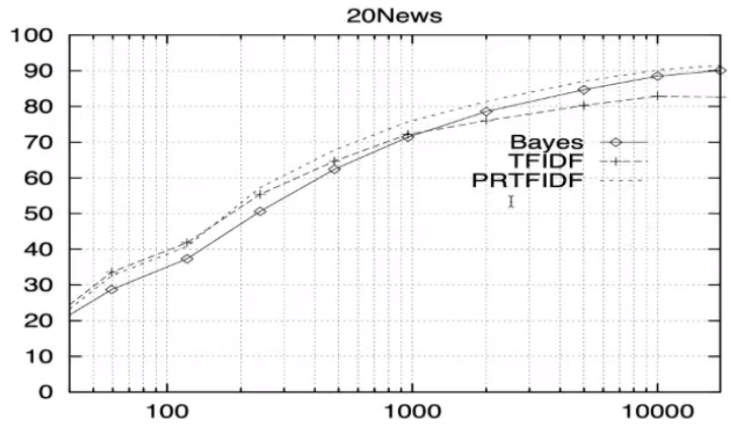
Another important assumption being made is the ordering of words given in the following

$$P(a_i = w_k|v_j) = P(a_m = w_k|v_j), \forall i, m$$

This means that whether the word is at index i or index m, we will treat it as the same. All we care about is the probability of the word, ignoring the word order.

Applying the standard Naive Bayes algorithm (Multinomial) we get an 89% classification accuracy with 20 class outputs and 1000 training documents.

It is worth noting the size of training set increases the accuracy of our classifier




Accuracy vs. Training set size (1/3 withheld for test)

## Zero Frequency problem

a generalisation of a bayesian estimate for our attribute is the following

$$\hat{P}(a_i|v_j) \leftarrow \frac{n_c + mp}{n + m}$$

Where the following variable are

- N : number of training examples where the class is equal to class we are checking
- Nc: number of examples where the class is equal AND the attribute matches
- P is the prior estimate P(ai | vj)
- M is the weight given to the prior

This is known as the m-estimate of probability.

Suppose however an attribute is missing.

- We can choose to ignore it and in training the instance is not included in frequency count for attribute-value-class combination
- In classification attribute is emitted from calculation

However we can do better.

Suppose we skimmed through an email, noticed the word lottery but didn't know if the word viagra was in the email. This makes a huge difference in classification as we don’t know whether to use the computation which includes viagra or not, as this will affect whether the outcome is spam/ham. To overcome this we can take the average of both cases.

$$P(Y|\text{lottery}) = P(Y|\text{Viagra} = 0, \text{lottery})P(\text{Viagra} = 0) + P(Y|\text{Viagra} = 1, \text{lottery})P(\text{Viagra} = 1)$$

Now in our example let's suppose that the word viagra only appears 10% of the time. This would give us the following calculation

$$P(Y = \text{spam}|\text{lottery} = 1) = 0.65 \cdot 0.90 + 0.40 \cdot 0.10 = 0.625$$
$$P(Y = \text{ham}|\text{lottery} = 1) = 0.35 \cdot 0.90 + 0.60 \cdot 0.10 = 0.375.$$

Because the occurrence of viagra is relatively are, the resulting distribution deviates only a little from the initial prediction that it is indeed just ham.

## Logistic Regression

We can use linear regression for classification in a certain way known as logistic regression.

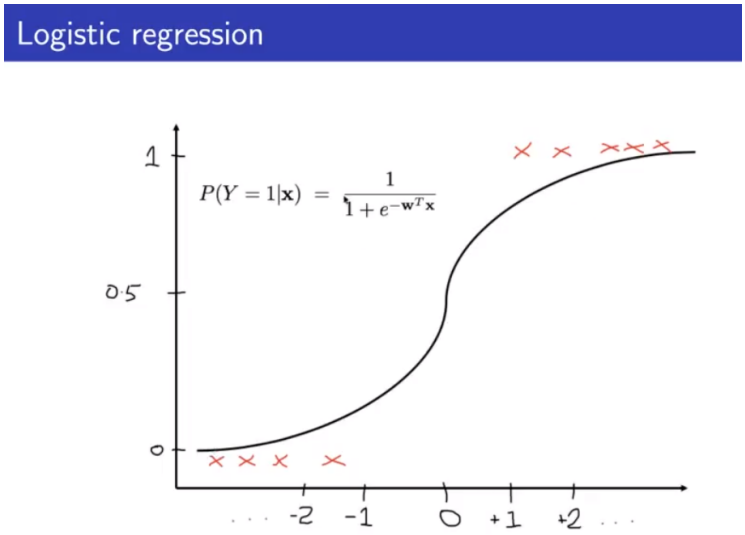
- Train a separate linear regression model for each class
    - Set y = 1 if example is in class
    - Set y = 0 if otherwise

When it comes to prediction

- For each example
    - Run through all regression models
    - Predict the class with the largest output value for y

This is known as multi-response linear regression, however we must be very careful as we do not obey the linear regression assumptions, so it may not always be suitable, however sometimes it works in practice.

Logistic regression then fits the sigmoid function



In the case of the two class classification problem if we model the probability of an instance being a positive example using the sigmoid function we get the following

$$P(Y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$$

Then we can compute the probability vs the alternative (1-P(a)) as the following

$$\ln \frac{P(Y = 1|\mathbf{x})}{1 - P(Y = 1|\mathbf{x})} = \mathbf{w}^T\mathbf{x}$$

The LHS of above is known as the logit and we are defining a linear model for our logit.

Unlike linear regression we have no analytical maximum likelihood solution to find weights w (in LR we used partial derivatives). This leaves us with the only alternative being gradient ascent to find a maximized log likelihood.

The (conditional) log likelihood is:

$$\sum_{i=1}^{N} y^{(i)} \log P(1|\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - P(1|\mathbf{x}^{(i)}))$$

Since y will either only have the value 0 or 1, it is more like a conditional if statement on each part of the addition, either we get

$$\sum_{i=1}^{N} y^{(i)} \log P(1|\mathbf{x}^{(i)})$$

When y = 1 (positive case), or we get

$$\sum_{i=1}^{N} [redacted] (1 - y^{(i)}) \log(1 - P(1|\mathbf{x}^{(i)}))$$

When y = 0 (negative case)

Over a set of N examples we choose the value of w that maximises this expression, and this generalises to multiple class versions (Y doesn't need to be binary).

## Summary of Classification 2

- We described the classification problem in machine learning
- We also outlined the issue of Inductive Bias
- Two major frameworks for classification were covered

  Distance-based. The key ideas are geometric.
  We discussed distance-based classification (Nearest Neighbour)

  Probabilistic. The key ideas are (mostly) Bayesian.
  We discussed generative (Naive Bayes) and discriminative (Logistic Regression) models
- So we have established a basis for learning classifiers
- Later we will see how to extend by building on these ideas

# Tree Learning

Trees are the single most popular data mining tool, they are

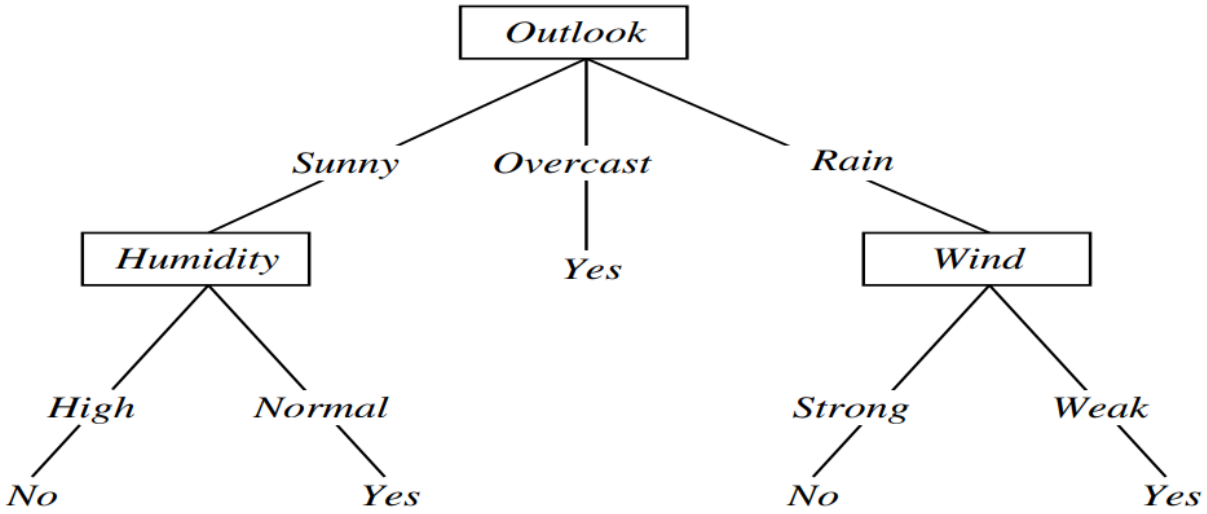
- Easy to understand
- Easy to implement
- Easy to use
- Computationally efficient to learn and run even on big data

They do however come with some drawbacks including

- High variance
- Overfitting

Their main purpose is to perform classification.

Here is an extremely simple example of a decision tree




## Structure

- Each internal node of the tree tests an attribute
- Branches correspond to the attribute value
- Each leaf represents a classification following down the path from root → leaf.

It is possible to represent the tree logically using boolean logical operators.

### Tree for AND and OR

$$X \wedge Y$$

```
X = t:
| Y = t: true
| Y = f: no
X = f: no
```

$$X \vee Y$$

```
X = t: true
X = f:
| Y = t: true
| Y = f: no
```

(⅔ true more complicated)

```
X = t:
| Y = t: true
| Y = f:
| | Z = t: true
| | Z = f: false
X = f:
| Y = t:
| | Z = t: true
| | Z = f: false
| Y = f: false
```

What we can see is that in general, decision trees represent a disjunction of conjunctions (a bunch of and statements or'd together) of constraints held on the attribute values instances.

## When to use decision trees?

- What can trees do?
    - With boolean values for instances X and class Y, the representation adopted by decision trees will allow us to represent our class Y as a boolean function of the X.
    - Given d input boolean variables there are 2^d possible input values, any specific function assigns Y = 1 to some subset, and Y = 0 to the rest
    - Any Boolean function can be trivially represented by a tree, all instances in that space are classified.
        - There are some functions where compact trees may not be possible (parity and majority functions, NP-complete)
        - Although it is possible in principle to express any boolean function, search and prior restriction may not allow us to find the correct tree in practice.
    - Trees are a re-representation of a truth table with 2^d rows. It is possible to compact your tree by taking into account what is common between one or more rows with the same output classification.
    - Trees are best used when we want readable models that combine logical tests with a probability based decision using a splitting method. A single decision tree can be a good start in this case
- When to use
    - Instances described by a mix of feature types (partly discrete, partly continuous)
        - Really common in real world

    - Target function is discrete valued (otherwise we use regression trees)
    - When we need a disjunctive hypothesis
        - A bunch of or statements stuck together (extremely useful for when we use sets of trees)
        - Rather than limiting to a single target function, we can say one or more hypothesis about our target function
    - When there is noisy data (deals well with noise)
    - Interoperabile models are required.
- Cases where trees have been used
    - Equipment or medical diagnosis
    - Credit risk analysis
    - Modeling calendar scheduling preferences
    - Many many more

# TDIDT (Top-Down Induction of Decision Trees)

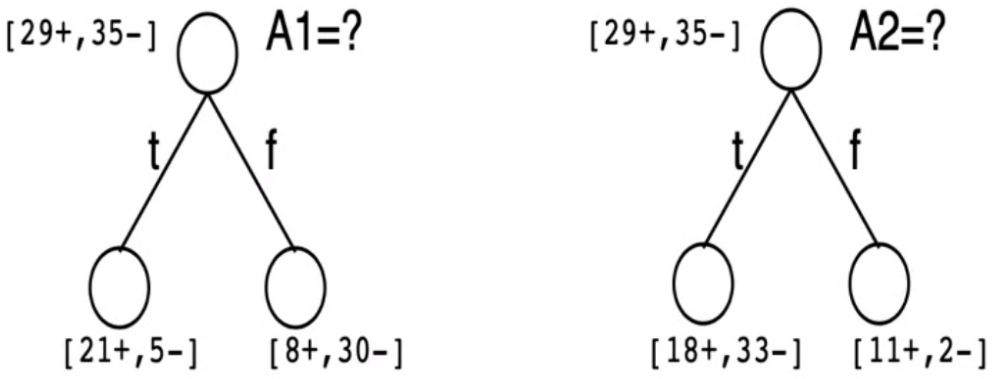
## ID3 (decision tree learning algorithm)

Simple loop to follow to partition data

1. A <- the best decision attribute for the next node (using a split criterion)
    a. Adding node to tree
2. Assign A as the decision attribute for our node in tree
3. For each value of A, create a branch with a new descendant node
4. Sort training examples to leaf nodes
5. If training examples are perfectly classified (no more splitting) then we stop, else we go back to step 1.

# Attribute splitting

Selecting the best attribute may not be so straight-forward, take for this example




How do we choose the best attribute for our split efficiently?

# Entropy

## Information encoding

Suppose we have a message consisting of a word constructed from 4 letters ABCD. now lets say that for the message we have the following probabilities

| | | | |
|---|---|---|---|
| $P(X=A)=\frac{1}{4}$ | $P(X=B)=\frac{1}{4}$ | $P(X=C)=\frac{1}{4}$ | $P(X=D)=\frac{1}{4}$ |

So it might make sense to encode the message with a 2 bit encoding (2 bits have 4 possible encodings).
So if we get message

BAACBADCDADDDA..

And encode with

A = 00, B = 01, C = 10, D= 11

We will transmit the following message

01000010010011101100111111100

This is the case when all probabilities are equal, however suppose that the probabilities are not equal, so we get the following probabilities

| | | | |
|---|---|---|---|
| $P(X=A)=\frac{1}{2}$ | $P(X=B)=\frac{1}{4}$ | $P(X=C)=\frac{1}{8}$ | $P(X=D)=\frac{1}{8}$ |

It is possible to create a coding for this transmission that uses only 1.75 bits on average with the following encoding

| | |
|---|---|
| A | 0 |
| B | 10 |
| C | 110 |
| D | 111 |

Note it may be possible to create multiple different encodings that have the same bit average, since our most frequent letter has the lowest number of encoded bits, we take less bits on average to encode.

Now Going back to our initial example with equal probabilities, suppose we have the following probability distributions

| | | |
|---|---|---|
| $P(X = A) = \frac{1}{3}$ | $P(X = B) = \frac{1}{3}$ | $P(X = C) = \frac{1}{3}$ |

In the normal case a naive encoding would cost 2 bits per symbol (in fact we could even fit an extra letter meaning we even waste potential space). Using the same approach as before, we can get a coding that costs only 1.6 bits per symbol on average.

| | |
|---|---|
| A | 0 |
| B | 10 |
| C | 11 |

Note it doesn't matter which letter gets the single bit since the probability distribution is even. The reason this gives us a smaller encoding is that on average
⅓ * 1 bit for A
⅓ * 2 bits for B
⅓ * 2 bits for C
If we sum it all together we get 5/3 ~~ 1.6 bits on average. This generalises to the following from information theory.
The optimal number of bits to encode a symbol with probability p is -log_2 p. So in this case the best we can do is -log_2 ⅓ for each of A, B and C, or 1.58 bits per symbol.

### Generalising

Suppose X can have m possible values each with probability distributions given as the following

| | | | |
|---|---|---|---|
| $P(X = V_1) = p_1$ | $P(X = V_2) = p_2$ | $\ldots$ | $P(X = V_m) = p_m$ |

Then the smallest number of bits on average per symbol needed to transmit a message X is the following

$$\begin{aligned} H(X) &= -p_1 \log_2 p_1 - p_2 \log_2 p_2 - \ldots - p_m \log_2 p_m \\ &= -\sum_{j=1}^{m} p_j \log_2 p_j \end{aligned}$$

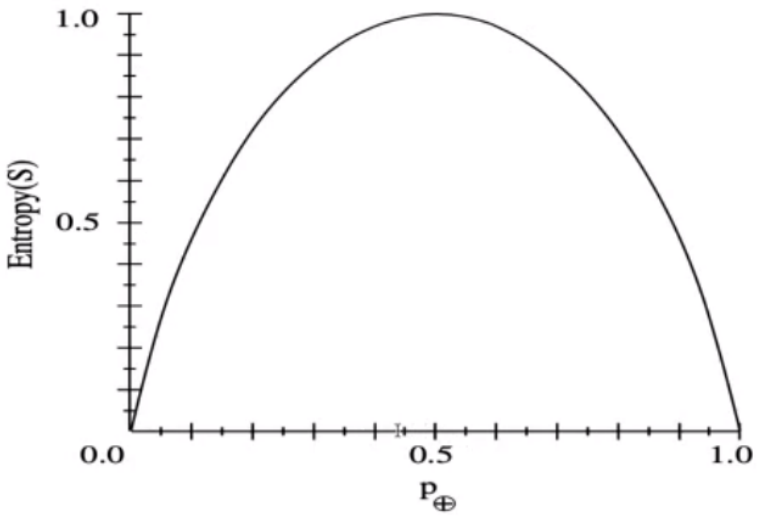
This gives us the entropy of X.

- High entropy correlates to uniform and boring data that doesn't give us a meaningful split (more chaotic)
- Lower entropy gives a varied and interesting split (less chaotic)

Entropy is the inverse of information, low entropy → more information, high entropy -> low information.

## Visualising Entropy

Consider this example for a simple 2-class distribution




The X axis shows the probability of classifying as positive, the y axis shows the entropy. What we can see is that at entropy = 1, we have a 50/50 split, which isn't meaningful in any way for splitting data, when entropy = 0 we have a perfect split 100/0 or 0/100 in both positive and negative cases. We always want to find the attribute that minimises entropy giving us more information and better splits.

Entropy is a measure of the impurity of a sample (purity means how pure it is in the split)

- Pure sample = all examples are of the same class (perfect split)

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

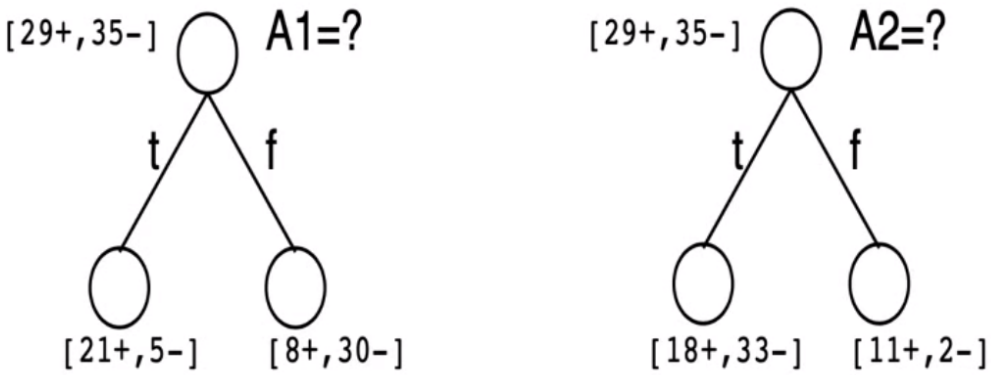
## Using Entropy to split data

We can measure information gain by splitting on an attribute A, by measuring the gain on sample S

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Where v is possible values of our attribute, and Sv is the subset of our sample that contains that attribute data. We don't event are for the value of the attribute using this split criterion, all we

care about is the effect the value of the attribute has on splitting the data. Applying this to our initial example of the split decision




We get the following reduction in entropy (gained information) for A1

$$
\begin{aligned}
Gain(S, A1) &= Entropy(S) - \left(\frac{|S_t|}{|S|}Entropy(S_t) + \frac{|S_f|}{|S|}Entropy(S_f)\right) \\
&= 0.9936 - \\
&= ((\frac{26}{64}(-\frac{21}{26}\log_2(\frac{21}{26}) - \frac{5}{26}\log_2(\frac{5}{26}))) + \\
&\quad (\frac{38}{64}(-\frac{8}{38}\log_2(\frac{8}{38}) - \frac{30}{38}\log_2(\frac{30}{38})))) \\
&= 0.9936 - ( 0.2869 + 0.4408 ) \\
&= 0.2658
\end{aligned}
$$

In a similar fashion for A2 we get the following

$$
\begin{aligned}
Gain(S, A2) &= 0.9936 - ( 0.7464 + 0.0828 ) \\
&= 0.1643
\end{aligned}
$$

What this tells us is that Attribute 1 gives us a better split as it lowers entropy.

## GINI Attribute Selection

A way to measure impurity more generally is to estimate class probability of a class k at node m in the following

$$\hat{p}_{mk} = \frac{|S_{mk}|}{|S_m|}.$$

Where |Smk| is the amount of examples in the sample at node m which have class output k, and |Sm| is just the size of the sample at the node. One way to classify at node m is by simply predicting the majority class, similar to KNN

$$\hat{p}_{mk}(m).$$

Pick the class with the most occurrences at the node. This also allows us to get misclassification error simply as

$$1 - \hat{p}_{mk}(m)$$

And now the entropy for K-class value sis the following

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

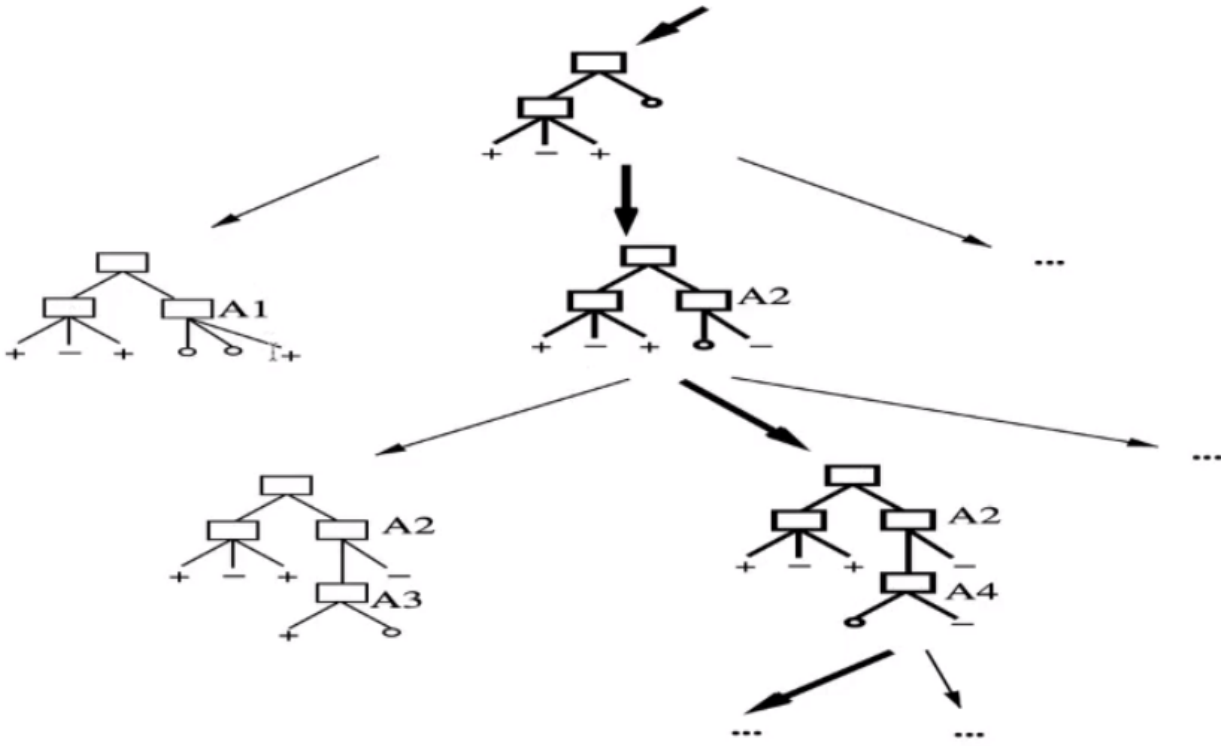
And for the CART split which uses “Gini Index” we have the following

$$\sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Where K is the set of all class outputs possible

## Greedy Search

The way we created our trees was through a greedy search which always splits on the attribute with the lowest entropy and forgetting all previous state, similar to a depth first search




We see in the diagram that we just follow the path down the lowest entropy. This however can be problematic as it won't always give us the best final tree, similar to greedy search won't always give an optimal solution but they will give it extremely fast.

This can be seen as a graph problem where each node in our graph is represented by a decision tree. For simplicity sake we consider only the two-class case, and all features being discrete giving us a binary tree for each node.

An edge connects two nodes if the trees differ in the following way: one of the leaf nodes in one vertex has been replaced by a non-leaf node testing a feature that has not been tested earlier with two leaves (new feature added to tree).

Despite fitting the full space of decision trees (under the binary assumption of our trees) we don't want all the trees, we only are looking for the smallest tree that represents our final state which is our target function that can fit the dataset.

ID3 uses a usual graph search technique of greedy search starting with the empty tree and greedily choosing the next state that results in the greatest increase in P(D|T) (greatest probability our data set will fit in the tree).

- This is done by taking the likelihood ratio of the old state and the new state and picking the new state out of all options with the highest likelihood ratio.

This gives us a set of trees (our path) with high posterior probabilities given D since we always chose the highest. We can now quantitatively answer questions such as P(y' = w1 |....) the probability of output class w1 given features, or we can even make a decision or a classification on an input data.

## Inductive Bias of Tree Learners

- Our Hypothesis space is complete
    - contains all finite discrete valued functions with respect to the attributes) one of which is our target function.
- Outputs one hypothesis
    - Choosing which one is the difficulty
- No back-tracking with greedy search
    - Falls victim to local minima problems.
- Statistically based search choices based on likelihoods.
    - Robust to noisy data

All these features our tree learners give us the inductive bias that we always prefer shorter trees. (go and find the shortest tree which predicts the data reasonably well). We will take the tree that goes up to a limit value of information gain.

We can choose to run for complete purity (each class is separated at leafs with 0 entropy) or we can have a relaxed decision rule e.g. 99 cases true 1 case false, can assume it will be false since entropy is close to 0.

Bias is a preference for some hypotheses rather than a restriction of the hypothesis space.

**This means we are performing an incomplete search of a complete hypothesis space**, rather than performing a complete search of an incomplete hypothesis space as seen with other learning models.

This inductive bias can be seen by Occam's Razor which says we always prefer the shortest hypothesis that fits the data.

### Occam's Razor

It is worth noting that Occam's razor mentions the shortest hypothesis that **FITS THE DATA**, a shit classifier is not suitable even if it's short.

- Entities should not be multiplied beyond necessity
- We don't need to go above and beyond simplicity is the best choice

We prefer shorter hypothesis for the following reasons

- Avoids overfitting
- There are fewer short hypothesis than long hypotheses
- Shorter hypothesis that fit data are not likely to be coincidences
    - Real world examples, when someone cuts straight to the chase it seems more likely, but if someone gives you a whole bunch of back story etc. it seems less likely.
    - Long hypothesis that fit data might be coincidences and we don't want that to be our model, leads to overfitting

However there are reasons why this may not be a good thing

- There are many ways to define small sets of hypothesis
    - E.g. all trees with a prime number of nodes that use attributes beginning with the letter "Z" and from mars, the most specific example possible.
- The argument requires a deeper formulation of complexity.

## Overfitting and how to avoid

Since trees are non parametric and can be expanded to the point where they can overfit easily this leads to common overfitting.

Greedy searches are prone to mistakes and suffer heavily from local minima, a choice earlier which may seem good may lead us to a worse overall solution than a sub-optimal choice earlier with a better solution in the end.

However we have a problem, we know that training error is an optimistic estimate of the true error of the model (your training error set can only tell you so much). This optimism increases as the training error decreases, if our training error is really low, we have a high optimism

- Suppose we have two models, h1 and h2 with training errors e1 and e2, and optimism o1 and o2.
    - The true error for h1 is given by Err1 = e1 + o1
    - The true error for h2 is given by Err2 = e2 + o2
- If e1 < e2 and Err1 > Err2, then we can say h1 has overfit
    - If model 1 has less error than model 2 for training
    - However in practice model 1 has more error than model 2
    - OVERFITTING
- This means a search method that is purely based on training data estimates may lead to overfitting the training data giving us a model that is not generalised.

To give a formal definition of overfitting

Consider error of hypothesis $h$ over

- training data: $error_{train}(h)$
- entire distribution $\mathcal{D}$ of data: $error_{\mathcal{D}}(h)$

**Definition**

Hypothesis $h \in H$ **overfits** training data if there is an alternative hypothesis $h' \in H$ such that

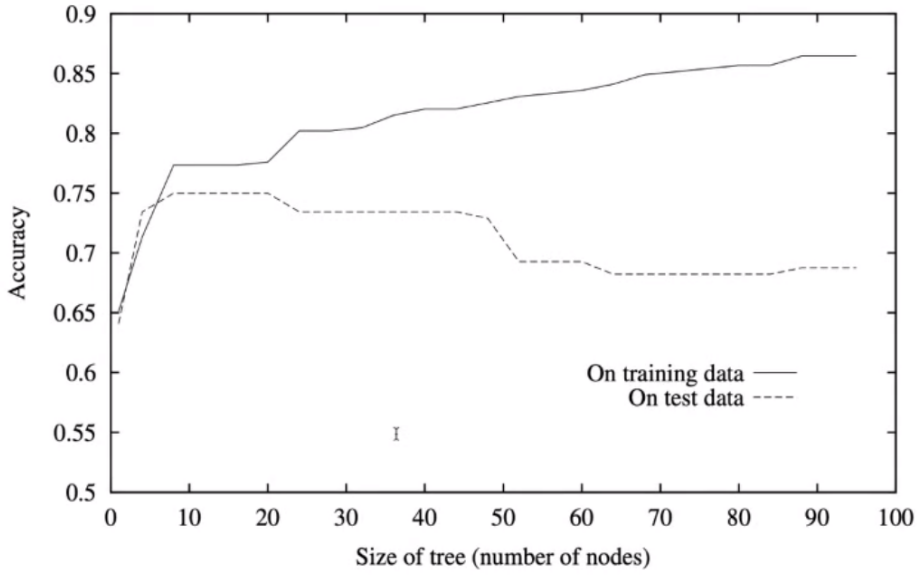
$$error_{train}(h) < error_{train}(h')$$

and

$$error_{\mathcal{D}}(h) > error_{\mathcal{D}}(h')$$

It is worth knowing that in a real world scenario we cannot get error on entire distribution of data, if we make an app predicting if someone is profitable to loan to, we don't know if we misclassified until the outcome which at that point the information is redundant. This error usually comes from some form of another sample that is hidden away during training, train-validation sets.

It is worth seeing the effect of overfitting on training and test data, the figure below shows how node increase impacts the accuracy in training and in testing




What we see is that on training as we increase the size of tree we reach much higher accuracy, but on testing this increase begins to fall off after a certain point, and this is due to overfitting.

Using our definition of overfitting we know that once we reach ~10 nodes, we have our training and test error almost the same, so we want to prune off larger trees which give us a large accuracy on training data, but on test data a much worse accuracy than smaller trees.

## Pruning

Pruning is a means of avoiding overfitting, with two types

- Pre pruning
    - Stop growing when data split is not statistically significant
    - Easier to implement than post-pruning
- Post pruning
    - Grow full tree and remove sub trees which overfit
    - Avoid the problem of early stopping that pre-pruning might have

To select the best tree

- Can measure performance over training data but this is not a good means as it will not be generalised
- Can measure performance over a separate validation data set made from training data, should always be done.
- MDL (not as efficient but very good approach)
    - Minimize the (size(tree) + size(misclassifications(tree)))

### Pre-Pruning

Can be based on a statistical threshold which stops growing the tree when there is no statistically significant association between any attribute and the class at a particular node.

In ID3, chi-squared test plus information gain

- Only statistically significant attributes were allowed to be selected by information gain procedure.

However this can be a problem

- As trees grow sample sizes at nodes get smaller meaning statistical tests become unreliable
- Pre-pruning may suffer from early stopping which stops the growth of a tree prematurely
- XOR/parity classic problem shows this
    - No individual attribute shows a significant association with the class (you need all to decide parity or XOR)
    - Target structure is only visible in fully expanded tree
    - Pre-pruning won't expand the root node as all features are meaningless splits
- Worth noting the XOR-type problems aren’t common in practice and pre-pruning is much faster than post-pruning

In a simple sense, we stop growing the tree when fewer than some lower bound on the number of examples at leaf. We set a cut-off point to where we stop splitting

- In c4.5 known as m parameter

- In sklearn the parameter is min_samples_leaf
- In sklearn we also have min_impurity_decrease which stops when we fall below the lower bound.

Pre-pruning a tree is good for use when we want to quickly make a tree that works fairly well to understand the data, however it should not be the final means. It does not maximise accuracy, however it does give us a sense of what the early most important features are

### Post Pruning

Build a full tree first as the first step of post pruning.

Problem

- How do we find in our training set the parts to prune off which are not generalised? Which parts are true to the training set and not the population?
- How can we see the effect of our pruning operations?

Pruning operations

- Subtree replacement
- Subtree replacement

There are many ways to determine whether we prune a node or not, this includes

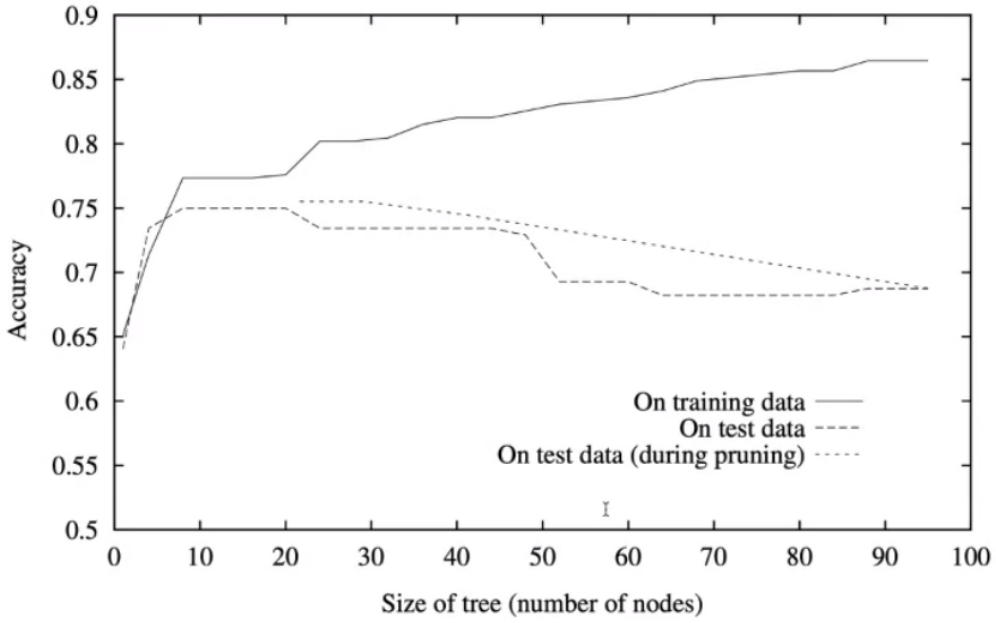
- Error estimation
- Significance testing
- MDL (Minimum description length) principle
- Reduced error pruning
- Error-based pruning

#### Reduced Error Pruning

1. Split our training set into a training and validation set
2. Keep pruning until further pruning is harmful
    a. Prune then evaluate the impact on the validation set of pruning each possible node.
    b. Greedily prune off a node and constantly re-evaluate until there are no more nodes to remove or until removing a node has a negative effect on validation set

This approach is good in that it will produce the smallest version of the most accurate subtree, however it does reduce the effective size of the training set.

Now if we look at the accuracy of pruning, we will see that it actually almost mimics the trend of the generalised accuracy




### Error Based Pruning

This method of pruning has many extensions, it uses both subtree-replacement and subtree raising. There is also a way to prune by converting the tree to rules.

#### *Subtree Replacement*

Tree is considered for replacement once all it's subtrees have been considered. This is a bottom-up approach, where we check from the bottom nodes first. Our goal here is to reduce error on unseen data only using data available in training set.

Our method of determining error is statistically motivated but not statistically valid. It uses the standard bernoulli process, and in practice it does work well. Our way to estimate error e now is given by the following

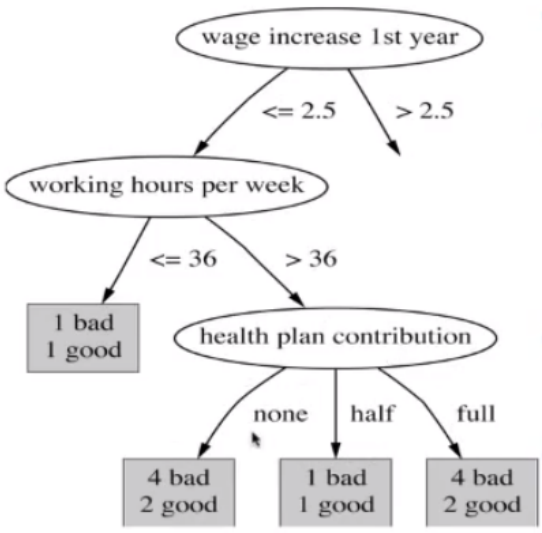
- Upper bound error estimate $e$ for a node (simplified version):

$$e = f + Z_c \cdot \sqrt{\frac{f \cdot (1-f)}{N}}$$

- $f$ is actual (empirical) error of tree on examples at the tree node
- $N$ is the number of examples at the tree node
- $Z_c$ is a constant whose value depends on *confidence* parameter $c$
- C4.5's default value for confidence $c = 0.25$
- If $c = 0.25$ then $Z_c = 0.69$ (from standardized normal distribution)

For our value c supplied to Zc, the direct correlation is that as c increases (more confident we are) Zc decreases (the error we are adding pessimistically). More confident == less chance of error.

Now here is an example of post-pruning using error-estimate




- health plan contribution: node measures $f = 0.36$, $e = 0.46$
- sub-tree measures:
  - none: $f = 0.33$, $e = 0.47$
  - half: $f = 0.5$, $e = 0.72$
  - full: $f = 0.33$, $e = 0.47$
- sub-trees combined $6 : 2 : 6$ gives $0.51$
- sub-trees estimated to give *greater* error so prune away

Since the health plan contribution node has less estimated error (0.46) than the combined subtrees (0.51) we decide to prune and replace.
Note this is not statistically valid but it works pretty well.

### Rule Post-Pruning

This method of pruning concerns the tree to an equivalent set of rules, and prunes each rule independently. It then sorts the final set of rules into a desired sequence of use. This is really

good if we are working with a simple classifier, also people prefer rules to trees for a model. However it does not scale well and is very slow for larger trees and datasets.

*Converting a Tree to rules*


IF $(Outlook = Sunny) \wedge (Humidity = High)$
THEN $PlayTennis = No$
IF $(Outlook = Sunny) \wedge (Humidity = Normal)$
THEN $PlayTennis = Yes$

Both identical.

Rules can be simpler than trees and maintain accuracy. A path from the root to leaf in an unpruned tree forms a rule (the whole tree is a set of rules). We can simplify rules independently by deleting conditions, we can generalise rules whilst maintaining accuracy.

This is a greedy rule simplification algorithm that

- Drops the condition giving lowest estimated error
- Continue while estimated error does not increase.

# Continuous Valued Variables

Decision trees originally were designed to work with discrete attributes only, however in practice we usually get a mix of discrete attributes and continuous attributes. So how do we deal with continuous values and numbers.
Consider for the example

- Temperature = 83
- We can create a threshold value to test a property on that attribute and create a discrete attribute set based off that
    - Threshold temperature > 70 → [T, F]

- Now this threshold gives us either true or false depending on the value of our continuous attribute.
- In practice continuous attributes have a binary split

This however raises the problem, how do we choose a threshold value?

## Choosing a threshold

Splits are evaluated on all possible points of splitting.

- Sort example on continuous attribute
- Find midway boundaries where class changes, for example in our temperature case here

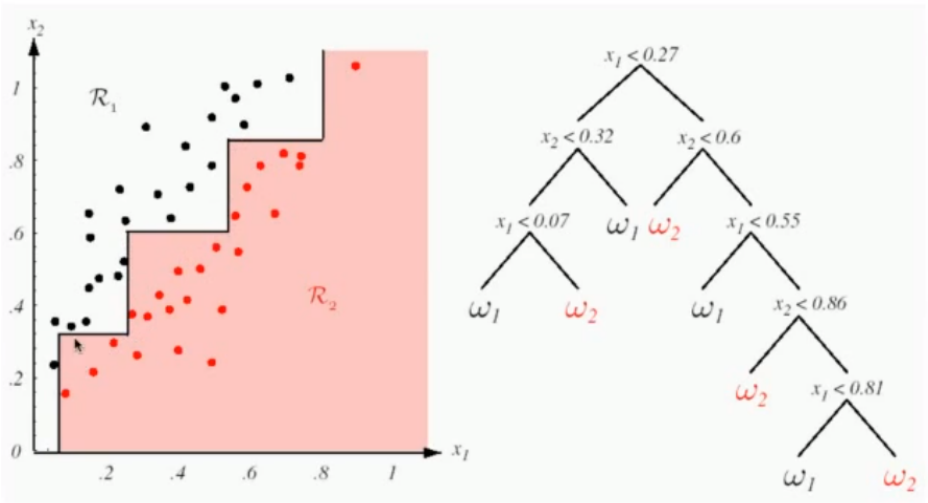
| | | | | | | |
|---|---|---|---|---|---|---|
| *Temperature*: | 40 | 48 | 60 | 72 | 80 | 90 |
| *PlayTennis*: | No | No | Yes | Yes | Yes | No |

- In this case, (48+60)/2 and (80+90)/2.
- Choose the best value by information gain (or entropy)
- Note that we want our split point to be in our data, so in the example above suppose that the 48-60 midpoint was the best split (48+60)/2 = 54, however this is not in our data, so what we can do is create our split point just after one of the boundaries, so 49 or 59.

There are two ways of splitting data

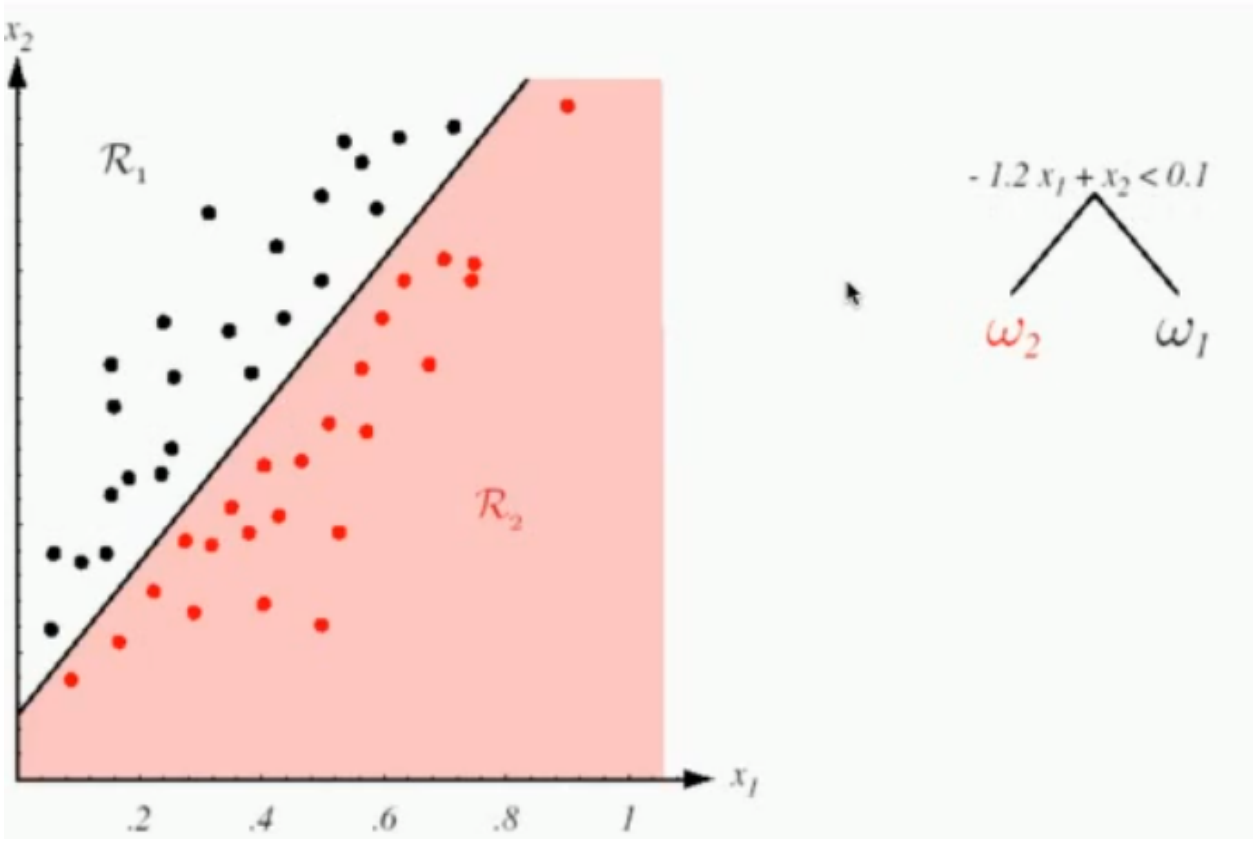
## Axis Parallel splitting

When we split on the threshold value of a feature, we will end up with a step-like function, where splits are made parallel to the axis, e.g. if we say split when x = 1, we will have a line at x = 1 forcing a split, this will end up with a step-like function as shown here



This is where all our splits here are based on the attribute value of x or y only, we can see a step-like function. This will do the job, however it may be more over-fitting vs a linear split. And we get a worse approximation of the data.

A better approach would be to split based on linear combination of features, shown below




However the draw being here is that it is more difficult to compute the linear combination in this case.

## Multi-valued discrete attributes

One problem of a discrete attribute that has multiple values is that it will be more preferred by our gain, as it will create more splits causing less entropy. It is more likely to split instances into pure subsets.

Imagine using Date = June 22, 2020 as an attribute, this causes many problems. For starters dates are historical so aren't likely to be used again.when comparing to other date attributes, multiple approaches. One is to split it into 3 categories (created attributes) however this isn't too useful for prediction. We will get high gain on the training set but for prediction it is useless.

To deal with this issue, we can instead use a GainRatio. This is to regularise our attribute splitting (for example in our date case).

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Here we can see split information is very similar to entropy, we are computing how many instances of each attribute split value occurs in our data set.

This gives us the entropy of S with respect to values of our attribute A.

- The information of how the partition of our attribute effects the dataset

This means that for attributes with more values, especially ones which are uniformly distributed across possible values, they will have a higher split information, decreasing the ratio.

## Working with costs

Some applications will require us to minimise costs or maximise costs, so now we want to make not only a good classifying tree, but one which minimises/maximises costs.

There are many approaches to this, one common one being evaluating information gain relative to costs. One example is the following formula for computing gain

$$\frac{Gain^2(S, A)}{Cost(A)}.$$

Here cost increase will decrease our gain, we are minimising costs in this case.

## Unknown Attribute values

In some cases examples have missing values for attributes. This is an area heavily researched and the following 3 approaches work well in decision tree learning

1. If node n tests A, assign most common value of A among other examples sorted to node n, (pick most occurring value of attribute)
2. Assign the most common value of A among other examples with the same target value. (picks the mode)
3. Assign probability pi to each possible value of our attribute vi
    a. Assign fraction pi of example to each descendant tree giving us partial examples.

## Windowing (memory issues)

Early implementations had training sets way too large to fit in memory. As a solution to this ID3 implemented windowing

1. Select subset of instances which is our window
2. Construct decision tree from the subset
3. Use the tree to classify instances not in the window
4. If all instances correctly classified we halt
5. Otherwise add selected misclassified instances to window and go to step 2

Windowing is still used commonly to this day, this is because it can improve accuracy on trees and relates heavily in ensemble learning.

## Regression Trees

An alternative approach to non-linear regression models. Similar to decision trees however

- Splitting criterion now minimizes intra-subset variation
- Pruning criterion now based on numeric error measure
- Leaf node predicts average class values of training instances reaching that node

Regression trees can help approximate piecewise constant functions in an easy to interpret way. Regression trees can work with discrete and continuous valued attributes.

### Variance Reduction

The variance of a boolean variable with success probability p is given by v = p(1-p) which is half the gini index. The goal of tree learning is to minimise the class variance in the leaves. In regression problems we define the variance in the following way

$$\mathrm{Var}(Y) = \frac{1}{|Y|} \sum_{y \in Y} (y - \overline{y})^2$$

If we split the set of target values Y into mutually exclusive sets {y1, y2, …, yn} then the weighted average variance can be given by

$$\text{Var}(\{Y_1, \ldots, Y_l\}) = \sum_{j=1}^{l} \frac{|Y_j|}{|Y|} \text{Var}(Y_j) = \ldots = \frac{1}{|Y|} \sum_{y \in Y} y^2 - \sum_{j=1}^{l} \frac{|Y_j|}{|Y|} \overline{y}_j^2$$

The first term before the minus is constant, so what we want to do is maximise what we are subtracting (the weighted average of squared means in the children).

## Example

Let's say we are given this data with a continuous valued output

| # | Model | Condition | Leslie | Price |
|---|---|---|---|---|
| 1. | B3 | excellent | no | 4513 |
| 2. | T202 | fair | yes | 625 |
| 3. | A100 | good | no | 1051 |
| 4. | T202 | good | no | 270 |
| 5. | M102 | good | yes | 870 |
| 6. | A100 | excellent | no | 1770 |
| 7. | T202 | fair | no | 99 |
| 8. | A100 | good | yes | 1900 |
| 9. | E112 | fair | no | 77 |

From this data we want to create splits in the data

$$\begin{aligned}
&\text{Model} = [\text{A100}, \text{B3}, \text{E112}, \text{M102}, \text{T202}] \\
&\qquad\qquad [1051, 1770, 1900][4513][77][870][99, 270, 625] \\
&\text{Condition} = [\text{excellent}, \text{good}, \text{fair}] \\
&\qquad\qquad [1770, 4513][270, 870, 1051, 1900][77, 99, 625] \\
&\text{Leslie} = [\text{yes}, \text{no}] \quad [625, 870, 1900][77, 99, 270, 1051, 1770, 4513]
\end{aligned}$$

From this data we get the following on variance.

The means of the first split are 1574, 4513, 77, 870 and 331, and the weighted average of squared means is $3.21 \cdot 10^6$. The means of the second split are 3142, 1023 and 267, with weighted average of squared means $2.68 \cdot 10^6$; for the third split the means are 1132 and 1297, with weighted average of squared means $1.55 \cdot 10^6$. We therefore branch on Model at the top level. This gives us three single-instance leaves, as well as three A100s and three T202s.

Now remember when we are computing variance we want to maximise the weighted average of squared means, so in this case we take the first split.

Next we split further based on model value.

For the A100s we obtain the following splits:

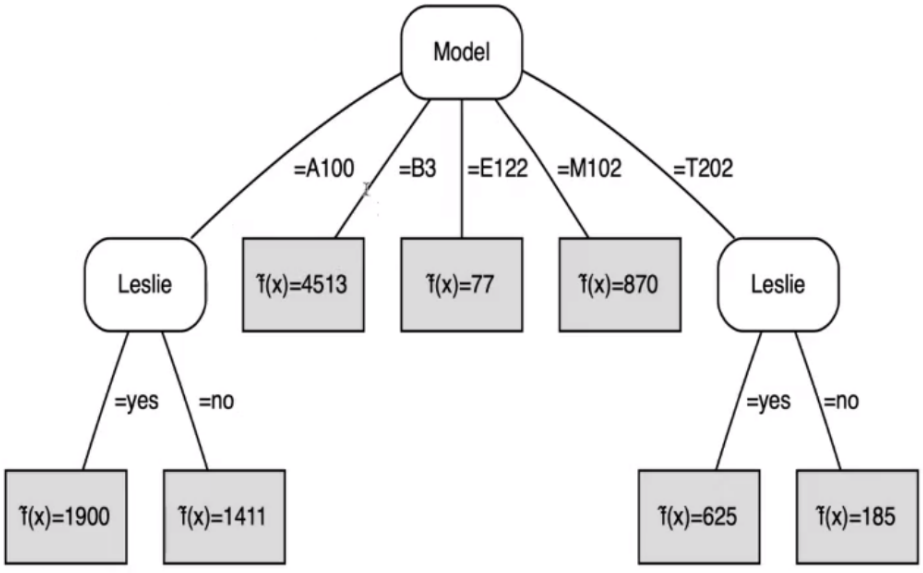
Condition = [excellent, good, fair] $[1770][1051, 1900][]$
Leslie = [yes, no] $[1900][1051, 1770]$

Without going through the calculations we can see that the second split results in less variance (to handle the empty child, it is customary to set its variance equal to that of the parent). For the T202s the splits are as follows:

Condition = [excellent, good, fair] $[][270][99, 625]$
Leslie = [yes, no] $[625][99, 270]$

Again we see that splitting on Leslie gives tighter clusters of values. The learned regression tree is depicted on the next slide.

This gives us our following regression tree

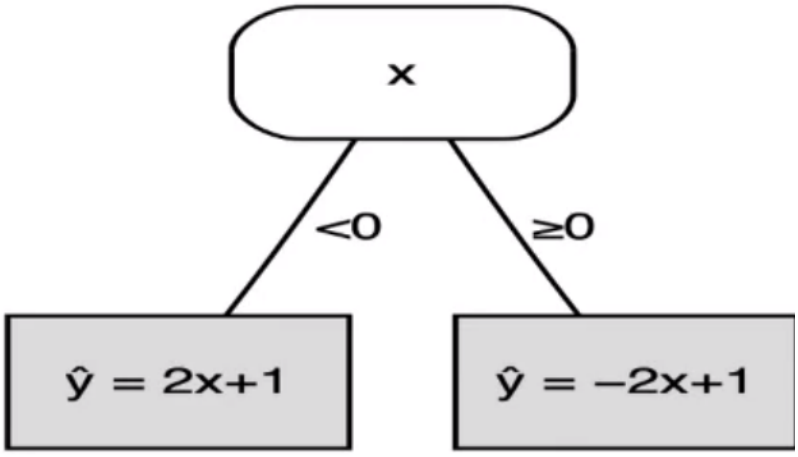


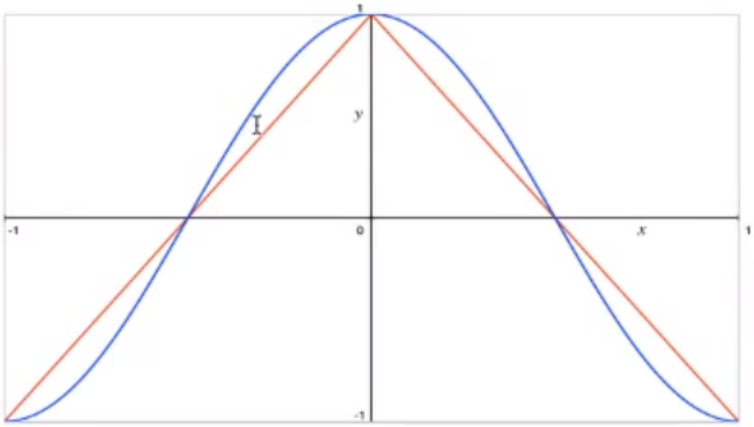
# Model trees

These are similar to regression trees but with linear regression functions at each node. Instead of a piecewise constant function we make piecewise linear functions.

Only a subset of the attributes is used for linear regression, those of which occur only in the subtree.

A very simple example of a small model tree would be to approximate y = cos(pi*x) on the interval -1 <= x <= 1.
Using a linear approximation is not of use since the best fit would be y = 0, but if we split the x axis into two intervals, we could find reasonable linear approximations on each interval. We will achieve this by using x both as a splitting feature and as a regression variable.




The leaf nodes have linear models. It wont fit perfectly but it's a good approximation.
Model trees are good since they don't just give a mean value, but instead use the input value of features to compute a continuous output value.

## Smoothing

The naive prediction method would be to just output the value of the LR model at the corresponding leaf node. We need to however be careful about fitting a linear model to a small amount of data, since each linear model works on subsets of data. To work around this we can use smoothing.

- Predicted value is weighted average of linear regression models along path from root to leaf

- Smoothing formula: $p' = \frac{np+kq}{n+k}$ where
  - $p'$ prediction passed up to next higher node
  - $p$ prediction passed to this node from below
  - $q$ value predicted by model at this node
  - $n$ number of instances that reach node below
  - $k$ smoothing constant
- Same effect can be achieved by incorporating the internal models into the leaf nodes

We control the constant k, so for example if a leaf has very low amount of data, we want more weighting on the previous nodes prediction, so we weigh higher q.

## Building the tree

- Splitting criterion is the standard deviation reduction

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i)$$

where $T_1, T_2, \ldots$ are the sets from splits of data at node.

Do this until either the standard deviation change becomes very unnoticeable (for example 5%), or until too few instances remain.

## Pruning the tree

Pruning is based on estimated absolute error of the linear regression models. Our heuristic estimate is given by the following

$$\frac{n+v}{n-v} \times \text{average\_absolute\_error}$$

Where n is the number of training instances that reach the node, and v is the number of parameters in the linear model.

Pruning is done by greedily removing terms to minimise the estimated error. Model trees allow heavy pruning where a single LR model can replace an entire subtree. Pruning proceeds bottom up similar to post-pruning. The error for the model at the node is compared to the error of the sub-tree.

### Discrete Values

To deal with discrete attributes, we can convert them to a binary attribute and treat them as a numeric value. These nominal values are then sorted using average class value for each one.

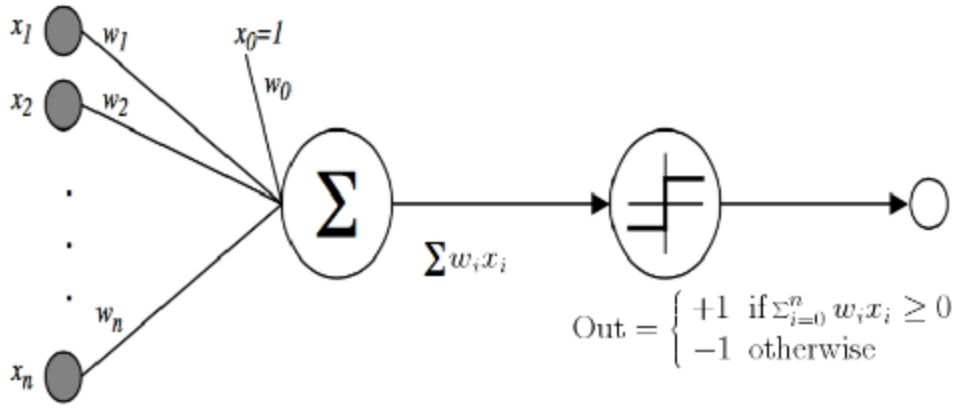
## Tree Summary

- Decision tree learning is a practical method for many tasks requiring classification, its in the top 10 most popular
- TDIDT family descended from ID3, searches complete hypothesis space for a solution, it is complete
- Uses a search bias, (inductive bias of tree) search for optimal tree (smallest tree)
- Overfitting is a common problem due to expressive hypothesis space and noisy data, so we must prune
- The common first method used in classification tasks, with a high pre-pruning variable can easily give us our top k features. Useful for a baseline.
- Trees are the first step, in understanding trees we can greatly improve them by using ensemble methods such as random forest.
- Regression trees were introduced in CART, allow us to predict a numeric value at the leaves
- Quinlan proposed a more powerful model tree inducer, which uses the value of features to predict the numeric value.
- Neural networks and model trees both do non-linear regressions.

# Neural Learning

## Perceptron

A linear classifier that can achieve perfect separation on linearly separable data is the perceptron, originally proposed as a simple neural networkbyF. Rosenblatt in the late 1950s. Originally implemented in software (based on the McCulloch-Pitts neuron from the 1940s), then in hardware as a 20x20 visual sensor array with potentiometers for adaptive weights.

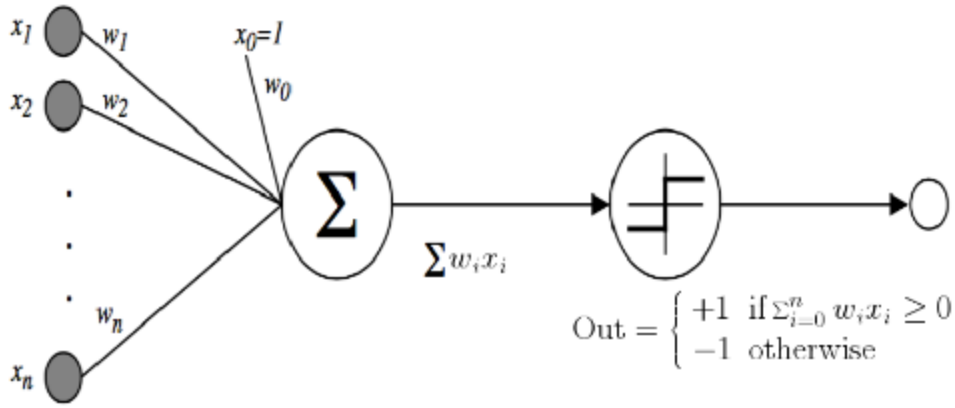



Output $o$ is thresholded sum of products of inputs and their weights:

$$o(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Xo is our bias/offset/intercept index, which isn't input but treated like one. It also has its own corresponding weight Wo. Sometimes we include it in the calculation, other times we don't, depending on the context.

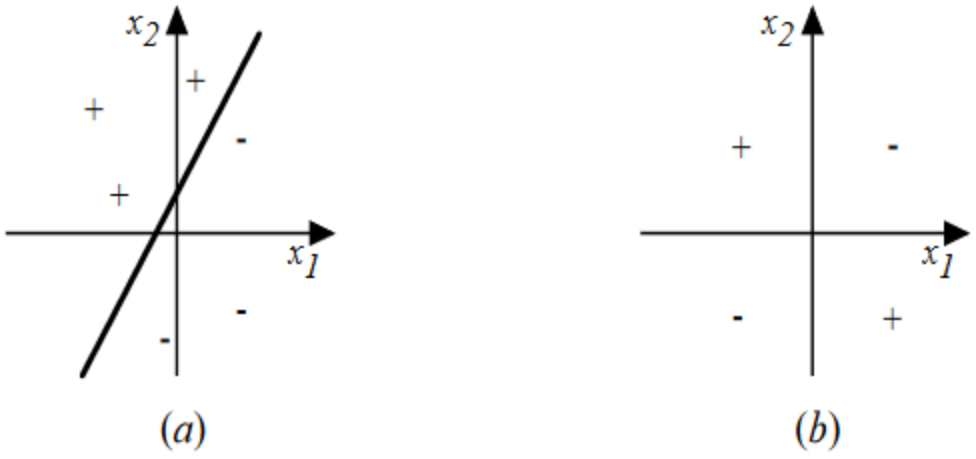
We take our output and pass it through a function (in the diagram it is a simple step function). This will give us our classification.




Or in vector notation:

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

## Decision Surface of a Perceptron




Represents some useful functions

- What weights represent $o(x_1, x_2) = AND(x_1, x_2)$?
- What weights represent $o(x_1, x_2) = XOR(x_1, x_2)$?

AND is possible while XOR is not. It is important that the function we represent is linearly separable.

## Perceptron Learning

Learning is “finding a good set of weights”

Perceptron learning is simply an iterative weight-update scheme:

$$w_i \leftarrow w_i + \Delta w_i$$

where the weight update ∆w_i depends only on misclassified examples and is modulated by a “smoothing” parameter η typically referred to as the“learning rate”.

The perceptron iterates over the training set, updating the weight vector every time it encounters an incorrectly classified example.

- For example, let $\mathbf{x}_i$ be a misclassified positive example, then we have $y_i = +1$ and $\mathbf{w} \cdot \mathbf{x}_i < t$. We therefore want to find $\mathbf{w}'$ such that $\mathbf{w}' \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$, which moves the decision boundary towards and hopefully past $x_i$.
- This can be achieved by calculating the new weight vector as $\mathbf{w}' = \mathbf{w} + \eta \mathbf{x}_i$, where $0 < \eta \leq 1$ is the *learning rate* (again, assume set to $1$). We then have $\mathbf{w}' \cdot \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x}_i + \eta \mathbf{x}_i \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$ as required.
- Similarly, if $\mathbf{x}_j$ is a misclassified negative example, then we have $y_j = -1$ and $\mathbf{w} \cdot \mathbf{x}_j > t$. In this case we calculate the new weight vector as $\mathbf{w}' = \mathbf{w} - \eta \mathbf{x}_j$, and thus $\mathbf{w}' \cdot \mathbf{x}_j = \mathbf{w} \cdot \mathbf{x}_j - \eta \mathbf{x}_j \cdot \mathbf{x}_j < \mathbf{w} \cdot \mathbf{x}_j$.

Note xi and xj are not the feature index’s at i and j, but rather the i’th and j’th training example, our weight is our vector of weights.

- The two cases can be combined in a single update rule:

$$\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$$

- Here $y_i$ acts to change the sign of the update, corresponding to whether a positive or negative example was misclassified
- This is the basis of the *perceptron training algorithm* for linear classification
- The algorithm just iterates over the training examples applying the weight update rule until all the examples are correctly classified
- If there is a linear model that separates the positive from the negative examples, i.e., the data is linearly separable, it can be shown that the perceptron training algorithm will converge in a finite number of steps.

## Perceptron training algorithm

**Algorithm** Perceptron$(D, \eta)$ // perceptron training for linear classification
**Input:** labelled training data $D$ in homogeneous coordinates; learning rate $\eta$.
**Output:** weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

```
w ←0 // Other initialisations of the weight vector are possible
converged←false
while converged = false do
    converged←true
    for i = 1 to |D| do
        if y_i w · x_i ≤ 0 then          // i.e., ŷ_i ≠ y_i
            w←w + η y_i x_i
            converged←false  // We changed w so haven't converged yet
        end
    end
end
```

## Perceptron Convergence

Perceptron training will converge (under some mild assumptions) for linearly separable classification problems

A labelled data set is linearly separable if there is a linear decision boundary that separates the classes

Assume:

Dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

At least one example in $D$ is labelled $+1$, and one is labelled -1.

$R = \max_i ||\mathbf{x}_i||_2$

A weight vector $\mathbf{w}^*$ exists s.t. $||\mathbf{w}^*||_2 = 1$ and $\forall i \; y_i \mathbf{w}^* \cdot \mathbf{x}_i \geq \gamma$

**Perceptron Convergence Theorem (Novikoff, 1962)**

*The number of mistakes made by the perceptron is at most* $(\frac{R}{\gamma})^2$.

$\gamma$ is typically referred to as the "margin".

### Decision Surface of a Perceptron

Unfortunately, as a linear classifier perceptrons are limited in expressivepower

So some functions not representable

- e.g., not linearly separable

For non-linearly separable data we'll need something else

However, with a fairly minor modification many perceptrons can be combined together to form one model

- multilayer perceptrons, the classic "neural network"

## Optimization

Studied in many fields such as engineering, science, economics, . . .

A general optimization algorithm:

1. start with initial point x = x_0
2. select a search direction **p**, usually to decrease f(x)
3. select a step length η
4. set s = η**p**
5. set x = x + s
6. go to step 2, unless convergence criteria are met

For example, could minimize a real-valued function: R^n -> R
Note: convergence criteria will be problem-specific.

Usually, we would like the optimization algorithm to quickly reach an answer that is close to being the right one.

- typically, need to minimize a function
  - e.g., error or loss
  - optimization is known as gradient descent or steepest descent
- sometimes, need to maximize a function
  - e.g., probability or likelihood
  - optimization is known as gradient ascent or steepest ascent

## When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g., raw sensor input)
- Output can be discrete or real-valued
- Output can be a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

Examples:

- Speech recognition (now the standard method)
- Image classification (also now the standard method)
- many others…

## Gradient Descent

To understand, consider simpler linear unit, where:

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's that minimize the squared error

$$E[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Gradient

$$\nabla E[\mathbf{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

Gradient vector gives direction of steepest increase in error E
Negative of the gradient, i.e., steepest decrease, is what we want

Training rule:

$$\Delta \mathbf{w} = -\eta \nabla E[\mathbf{w}]$$

I.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \mathbf{w} \cdot \mathbf{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

GRADIENT-DESCENT($training\_examples, \eta$)

*Each training example is a pair $\langle \mathbf{x}, t \rangle$, where $\mathbf{x}$ is the vector of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).*

Initialize each $w_i$ to some small random value

Until the termination condition is met, Do

  Initialize each $\Delta w_i$ to zero

  For each $\langle \mathbf{x}, t \rangle$ in $training\_examples$, Do

    Input the instance $\mathbf{x}$ to the unit and compute the output $o$

    For each linear unit weight $w_i$

      $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$

  For each linear unit weight $w_i$

    $w_i \leftarrow w_i + \Delta w_i$

## Training Perceptron vs. Linear unit

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

## Incremental (Stochastic) Gradient Descent

**Batch mode** Gradient Descent:
Do until satisfied

- Compute the gradient $\nabla E_D[\mathbf{w}]$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

**Incremental mode** Gradient Descent:
Do until satisfied

- For each training example $d$ in $D$
  - Compute the gradient $\nabla E_d[\mathbf{w}]$
  - $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

Batch:

$$E_D[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Incremental:

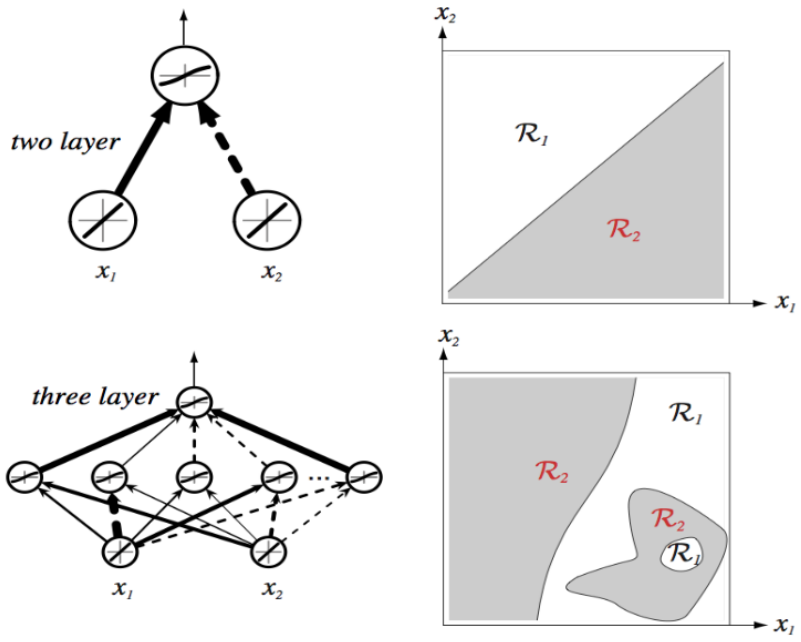
$$E_d[\mathbf{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental or Stochastic Gradient Descent (SGD) can approximate Batch Gradient Descent arbitrarily closely, if η made small enough

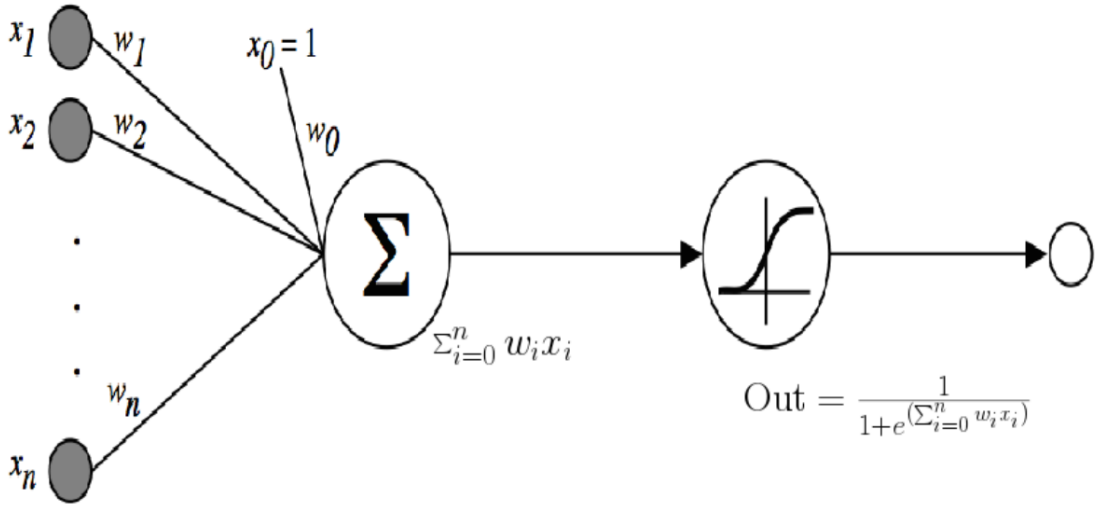
Very useful for training large networks, or online learning from data streams

Stochastic implies examples should be selected at random

## Multilayer Networks of Sigmoid Units




## Sigmoid Unit




Same as a perceptron except that the step function has been replaced by a smoothed version, a sigmoid function.

Note: in practice, particularly for deep networks, sigmoid functions are less common than other non-linear activation functions that are easier to train,but sigmoids are mathematically convenient.

Why use the sigmoid function $\sigma(x)$ ?

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Start by assuming we want to minimize squared error $\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$ over a set of training examples $D$.

Error Gradient for a Sigmoid Unit

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right) \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

## Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied, Do

For each training example, Do

Input the training example to the network and compute the network outputs

For each output unit $k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

For each hidden unit $h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

Update each network weight $w_{ji}$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

## More on Backpropagation

A solution for learning highly complex models . . .

- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Can learn probabilistic models by maximising likelihood

Minimizes error over all training examples

- Training can take thousands of iterations → slow!
- Using network after training is very fast

Will converge to a local, not necessarily global, error minimum

- May be many such local minima
- In practice, often works well (can run multiple times)
- Often include weight momentum α

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- Stochastic gradient descent using “mini-batches”

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Models can be very complex

- Will the network generalize well to subsequent examples?
  - may underfit by stopping too soon
  - may overfit. . .

Many ways to regularize network, making it less likely to overfit

- Add term to error that increases with magnitude of weight vector

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Other ways to penalize large weights, e.g., weight decay
- Using ”tied” or shared set of weights, e.g., by setting all weights to their mean after computing the weight updates
- Many other ways . . .

## Expressive Capabilities of ANNs

Boolean functions:

- Every Boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

Being able to approximate any function is one thing, being able to learn it is another . . .

## Neural networks for classification

Sigmoid unit computes output $o(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x})$

Output ranges from $0$ to $1$

Example: binary classification

$$o(\mathbf{x}) = \begin{cases} \text{predict class } 1 & \text{if } o(\mathbf{x}) \geq 0.5 \\ \text{predict class } 0 & \text{otherwise.} \end{cases}$$

Questions:

- what error (loss) function should be used ?
- how can we train such a classifier ?

Minimizing square error (as before) does not work so well for classification

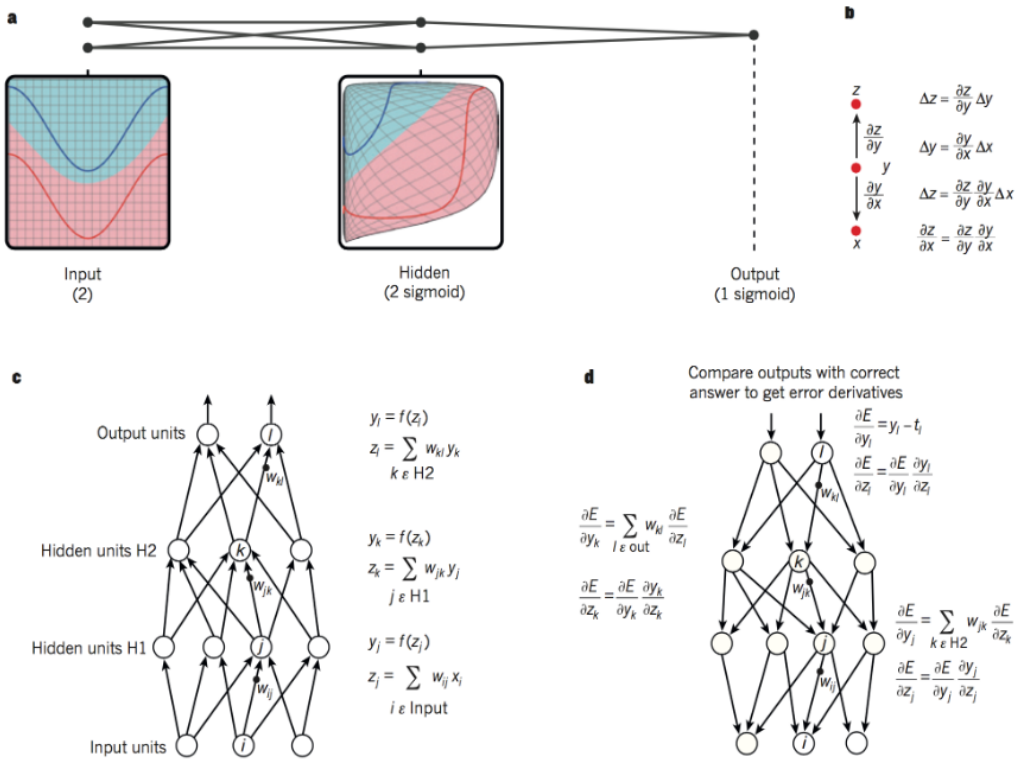
If we take the output o(x) as the probability of the class of x being 1, the preferred loss function is the cross-entropy

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

where:
td∈{0,1} is the class label for training example d, and o_d is the output of the sigmoid unit, interpreted as the probability of the class of training example d being 1.

To train sigmoid units for classification using this setup, can usegradientascentwith a similar weight update rule as that used to train neural networks by gradient descent – this will yield the maximum likelihood solution.

# Deep Learning




## Architectures

Most successful deep networks do not use the fully connected network architecture we outlined above.

Instead, they use more specialised architectures for the application of interest (inductive bias).

Example: Convolutional neural nets (CNNs) have an alternating layer-wise architecture inspired by the brain's visual cortex. Works well for image processing tasks, but also for applications like text processing.

Example: Long short-term memory (LSTM) networks have recurrent network structure designed to capture long-range dependencies in sequential data, as found, e.g., in natural language (although now often superseded by transformer architectures).

Example:  Autoencoders are a kind of unsupervised learning method.They learn a mapping from input examples to the same examples as output via a compressed (lower dimension) hidden layer, or layers.

## Activation Functions

Problem:  in very large networks, sigmoid activation functions cansaturate,i.e., can be driven close to 0 or 1 and then the gradient becomes almost 0 – effectively halts updates and hence learning for those units.

Solution: use activation functions that are non-saturating., e.g., “RectifiedLinear Unit” or ReLu, defined as f(x) = max(0,x).

Problem: sigmoid activation functions are not zero-centred, which can cause gradients and hence weight updates become “non-smooth”.

Solution:  use zero-centred activation function, e.g.,tanh, with range[−1,+1].  Note that tanh is essentially a re-scaled sigmoid.

Derivative of a ReLu is simply

$$\frac{\partial f}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases}$$

## Regularization

Deep networks can have millions or billions or parameters.
Hard to train, prone to overfit.
What techniques can help ?
Example:  dropout

- for each unit u in the network, with probability p, “drop” it, i.e.,ignore it and its adjacent edges during training
- this will simplify the network and prevent overfitting
- can take longer to converge
- but will be quicker to update on each epoch
- also forces exploration of different sub-networks formed by removing of the units on any training run

# Kernel Methods

The following outlines scenarios for predictive machine learning

| *Task* | *Label space* | *Output space* | *Learning problem* |
|---|---|---|---|
| Classification | $\mathcal{L} = \mathcal{C}$ | $\mathcal{Y} = \mathcal{C}$ | learn an approximation $\hat{c} : \mathcal{X} \to \mathcal{C}$ to the true labelling function $c$ |
| Scoring and ranking | $\mathcal{L} = \mathcal{C}$ | $\mathcal{Y} = \mathbb{R}^{\lvert\mathcal{C}\rvert}$ | learn a model that outputs a score vector over classes |
| Probability estimation | $\mathcal{L} = \mathcal{C}$ | $\mathcal{Y} = [0,1]^{\lvert\mathcal{C}\rvert}$ | learn a model that outputs a probability vector over classes |
| Regression | $\mathcal{L} = \mathbb{R}$ | $\mathcal{Y} = \mathbb{R}$ | learn an approximation $\hat{f} : \mathcal{X} \to \mathbb{R}$ to the true labelling function $f$ |

## Scoring

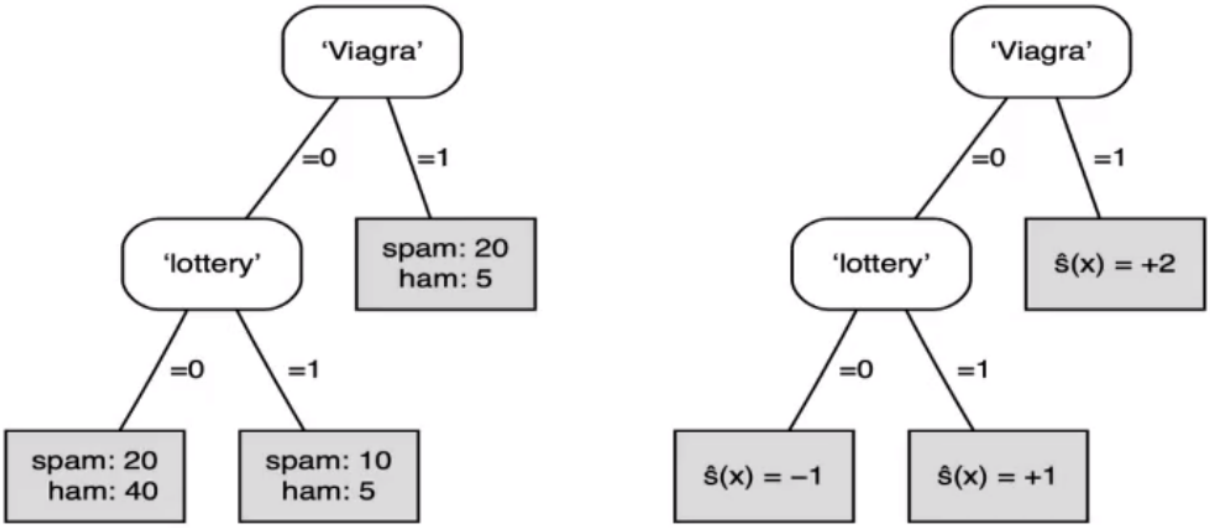
A classifier is a mapping from inputs to a set of discrete finite outputs, usually which is a small set of class outputs. Learning a classifier will require us to construct our function such that it matches the true function fitting as closely as possible, not on the training set only but also in a general sense.

A scoring classifier works differently in that it maps the inputs into a vector (size of number of classes) of real numbers that represent the scoring and ranking.

The boldface notation indicates that a scoring classifier outputs a vector $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \ldots, \hat{s}_k(x))$ rather than a single number; $\hat{s}_i(x)$ is the score assigned to class $C_i$ for instance $x$.

So each element of s(x) is the class score for that element.This score indicates how likely it is that the class label applies to our input.

Example of a decision tree using scoring method




Note positive implies spam (2 class problem) sign indicates prediction, magnitude indicates confidence. In this example we just took the ratio of spam/ham and took the log to get the score.

## Margins and loss functions

If we take the true class c(x) as positive for positive examples, and negative for negative examples, we can then quantify z(x) as c(x) * s(x) where s(x) is our strength of our example (Score). This quantity is known as the margin assigned by the scoring classifier.

We can use a loss function to reward large positive margins, and penalise large negative values.

$L : \mathbb{R} \mapsto [0, \infty)$ which maps each example's margin $z(x)$ to an associated loss $L(z(x))$.

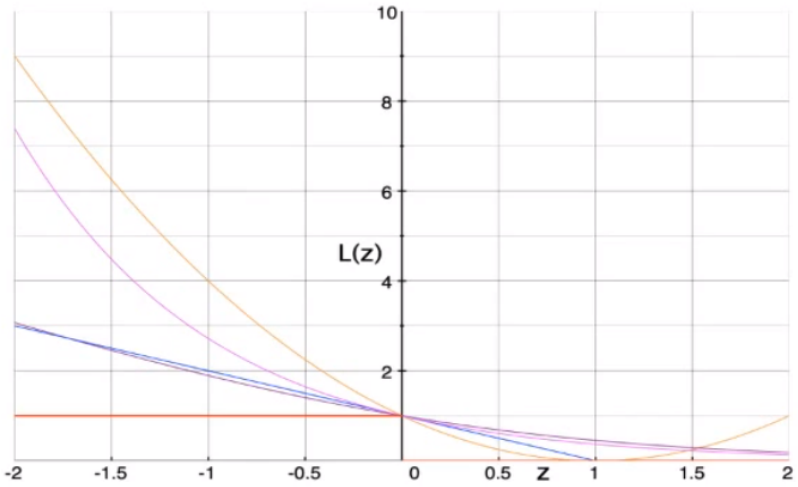
We will assume for our loss function that L(0) = 1, which is caused by having the example on the decision boundary. Furthermore for our function
L(z) >= 1 for z < 0 and usually also 0 <= L(z) < 1 for z > 0
Increase the loss when we make a mistake, and when we don't make a mistake our loss is between 0 and 1.

To compute the average loss over a test set we use the following example where Te is the test set

$$\frac{1}{|Te|} \sum_{x \in Te} L(z(x)).$$

Taking mean of loss on training set

## Loss Functions



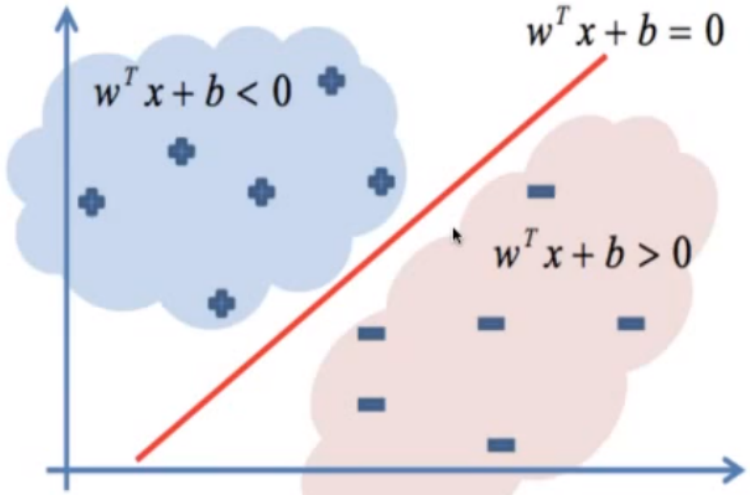

From bottom-left: (i) 0–1 loss $L_{01}(z) = 1$ if $z \leq 0$, and $L_{01}(z) = 0$ if $z > 0$; (ii) hinge loss $L_{\mathrm{h}}(z) = (1 - z)$ if $z \leq 1$, and $L_{\mathrm{h}}(z) = 0$ if $z > 1$; (iii) logistic loss $L_{\log}(z) = \log_2(1 + \exp(-z))$; (iv) exponential loss $L_{\exp}(z) = \exp(-z)$; (v) squared loss $L_{\mathrm{sq}}(z) = (1 - z)^2$ (can be set to 0 for $z > 1$, just like hinge loss).

1. Simple step function, value is either 0 or 1 depending if we get classification right or wrong
2. Hinge function, similar to step but it uses (1-z) if z <= 1 which basically says, it's not good enough just to get the right classification but we also need to be confident in it (higher z) to reduce loss.
3. Hinge loss will be main focus

# Linear classification

We can create a two-class classifier which separates instances into separate groupings shown below.

$$f(x) = sign(w^T x + b)$$




Here we have a line that splits the positive and negative examples based on the sign of our function. Our aim here is to define a decision boundary with a hyperplane in feature space. A difficulty in this is choosing a decision boundary, often the gradient/intercept being different vastly changes classification in a general sense.

## Separation

There are different techniques used by different algorithms

- Basic linear classifier finds the class means (centroids) and joins them by a straight line, the perpendicular bisector of this line is the separating hyperplane (SEE KNN NOTES)
- Nearest Neighbour (or weighted NN)
- Naive Bayes
- Logistic regression
- Perceptron training which uses iterative reweighting with gradient descent
    - Depending on starting conditions it may find different models

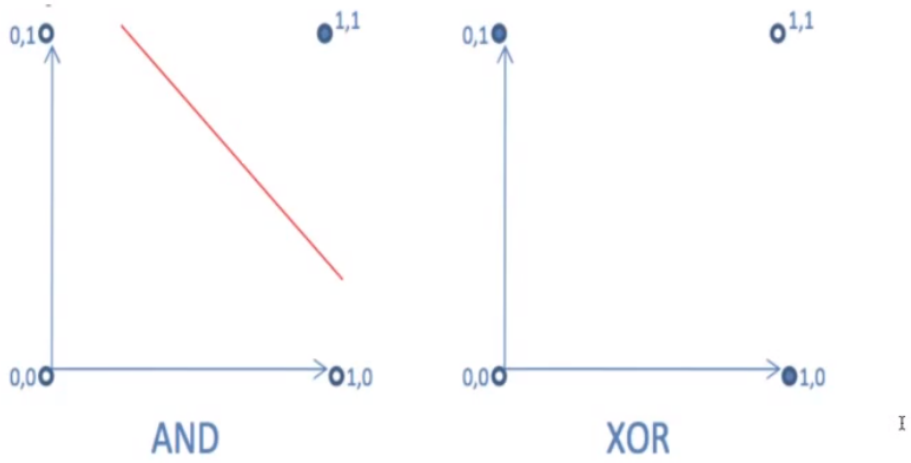
However none of these methods have explicitly told us how to find the hyperplane for separation, they only classify. We want to find an optimal linear classification learning method.

One approach is to define the empirical risk (error) on the data and computing the maximum margin separating hyperplanes. This will minimise the empirical risk. Our goal is to minimise empirical risk whilst also making sure our model doesn't get too complex which leads to

overfitting. Under Vapnik's framework for statistical learning we have a way to structurally minimise risk.

## Issues in linear classification

Some data cannot be linearly separated by a hyperplane, take for example XOR, AND can be separated, but we cannot linearly separate XOR




Linear classifiers can't model non-linear class boundaries. One way around this is Non-linear mapping where we map attributes into new space consisting of combinations of attribute values, this is feature construction or also called basis expansion.
An example would be, all products with n factors that can be constructed from the attributes

Suppose for 2 attributes z1 and x2, we have all products with n = 3 factors, giving us a new feature space, giving us a new linear space (note we can just say x1^3 = some new linear variable) to maintain linearity. In new space it is linear.

$$y = w_1 x_1^3 + w_2 x_1^2 x_2 + w_3 x_1 x_2^2 + w_4 x_2^3$$

There are three big drawbacks to doing this however

1. With 10 attributes and n = 5, we have to learn more than 2000 weights (combinatorially complicated)
2. Linear regression (with attribute selection) running time is O(n^3) where n is the number of attributes
3. Overfitting, when it becomes too nonlinear and the number of coefficients are large relative to the number of training instances, we begin overfitting.
4. Curse of dimensionality applies here as well

## Dual, Example Linear Classification

Everytime an example Xi is misclassified we add YiXi to the weight vector (similar to perceptron). After training is completed, each example has been misclassified zero or more times. We can say the number of misclassified examples as alpha for our example Xi. This allows us to express our weight vector as.

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

This gives us a new perspective to learning, up until now we think how does the model change itself to fit the training set. This approach however similar to KNN is fully just data reliant.

In the dual (opposite), instance based view of linear classification we learn instance weights rather than feature weights.

$$\hat{y} = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}\right)$$

Here we learn weights on the data (instances) to get our model , whereas before, we learn weights on the attributes.

This means during training, all we need is the Gram Matrix, where it is just all the pairwise dot products of our training data.

$$\text{the } n\text{-by-}n \text{ matrix } \mathbf{G} = \mathbf{X}\mathbf{X}^{\mathrm{T}}$$

Where X is our training data.

Perceptron training using our dual form

**Algorithm** DualPerceptron($D$) // perceptron training in dual form
**Input:** labelled training data $D$ in homogeneous coordinates
**Output:** coefficients $\alpha_i$ defining weight vector $\mathbf{w} = \sum_{i=1}^{|D|} \alpha_i y_i \mathbf{x}_i$
$\alpha_i \leftarrow 0$ for $1 \leq i \leq |D|$
$converged \leftarrow$ false
**while** $converged =$ false **do**
  $converged \leftarrow$ true
  **for** $i = 1$ to $|D|$ **do**
    **if** $y_i \sum_{j=1}^{|D|} \alpha_j y_j \mathbf{x}_i \cdot \mathbf{x}_j \leq 0$ **then**
      $\alpha_i \leftarrow \alpha_i + 1$
      $converged \leftarrow$ false
    **end**
  **end**
**end**

In here it is extremely similar to the perceptron algorithm, we can also notice in the if statement is is the equivalent of wixi * xi, where our wi is simply just the weight summed over the entire training set, and i is just the instance of data we check, we do this till we converge.

## Support Vector Machine

SVM's are a linear classifier which learn decision boundaries by maximising the margin. SVM's are powerful in avoiding overfitting as they learn a form of decision boundary known as the maximum margin hyperplane. They are fast for mapping to non linear spaces, they use a mathematical trick to avoid the creation of pseudo-attributes in transformed instance space. The non-linear space is only created implicitly, not set in memory.

### Kernel Trick

This is the mathematical trick that implicitly creates our new feature space.

Let $\mathbf{x}_1 = (x_1, y_1)$ and $\mathbf{x}_2 = (x_2, y_2)$ be two data points, and consider the mapping $(x, y) \mapsto (x^2, y^2, \sqrt{2}xy)$ to a three-dimensional feature space. The points in feature space corresponding to $\mathbf{x}_1$ and $\mathbf{x}_2$ are $\mathbf{x}'_1 = (x_1^2, y_1^2, \sqrt{2}x_1y_1)$ and $\mathbf{x}'_2 = (x_2^2, y_2^2, \sqrt{2}x_2y_2)$. The dot product of these two feature vectors is

$$\mathbf{x}'_1 \cdot \mathbf{x}'_2 = x_1^2x_2^2 + y_1^2y_2^2 + 2x_1y_1x_2y_2 = (x_1x_2 + y_1y_2)^2 = (\mathbf{x}_1 \cdot \mathbf{x}_2)^2$$

Note the mapping was chosen by the user. What we can observe by computing the dot product in our new feature space is that it is identical to performing the dot product on our initial datapoint and squaring the result. Now we are actually obtaining the dot products in the new feature space without making the new feature vectors. A function that computes the dot product in feature space directly from the vectors in the original space is called a kernel. In this case our kernel here is seen as

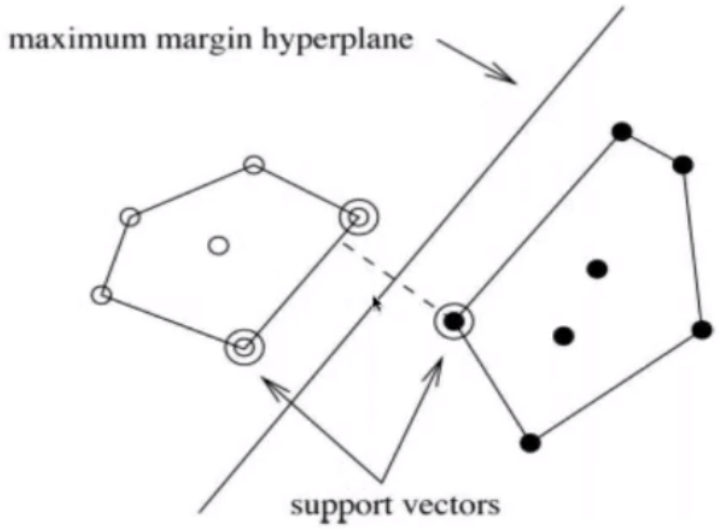
$$\boldsymbol{\kappa}(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 \cdot \mathbf{x}_2)^2.$$

## Training SVMS

For a linearly separable two class data set, the maximum margin hyperplane is the classification surface which

1. Correctly classifies all examples in the data set
2. Has the greatest separation between classes

The convex hull of instances in each class is the tightest enclosing convex polygon which captures all instances with the class values. For a linearly separable two class data set, convex hulls will not overlap. The maximum margin hyperplane is the perpendicular line to the shortest line connecting the two convex hulls at the halfway point. The more separated the classes, the larger the margin meaning more generalisation.



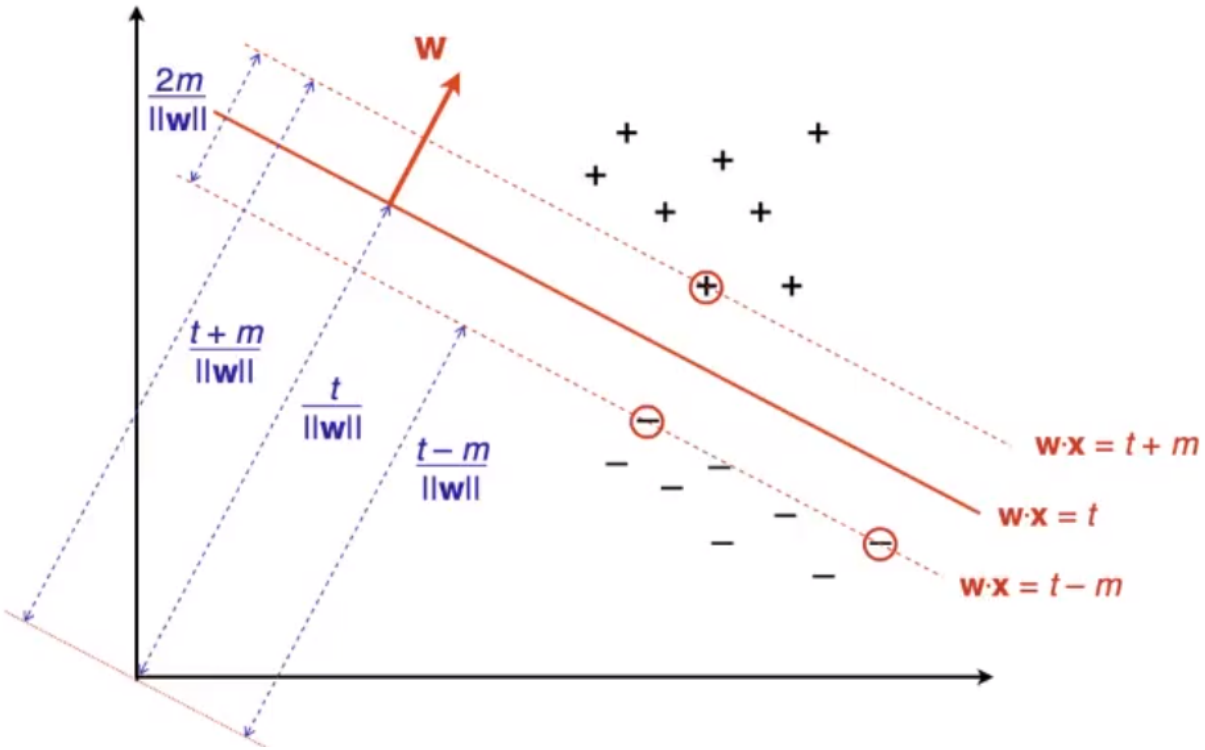

Note this method is susceptible to noise.

## Support Vectors

- These are the instances closest to the maximum margin hyperplane
- Important to note that these instances define maximum margin hyperplane, all other instances are irrelevant once fit and won't change the final classifier, this is because once we have figured out what the support vectors are, we have already got our classifier.

Under the assumption that you can linearly separate the classes, we can create a method to determine these support vectors. This is a constrained quadratic optimisation problem which has standard algorithms or more specific built for purpose algorithms to solve, e.g. SMO or LibSVM.

The figure below shows the geometry behind a SVC




We can observe that the line is determined by our weight vector. We notice that the distance from the support vector to the line separating is m/||w||. In order to find the best most generalised classifier, we want to find the boundary that maximises this value m/||w|| giving us the largest margin.
To do this we have 2 options. We can either increase m, or decrease w. In practice we treat m as 1, and rather decrease our weight vector instead.

From the diagram we can also observe that the support vectors lie on the parallel line that is translated by + and - m.

Since we are free to rescale t, ||w|| and m, it is custom to always choose m = 1, and minimise ||w|| which is equivalent and more convenient to represent as ½||w||^2. This all relies on the fact that training points don't fall inside the margin.

This leads up to our quadratic constrained optimisation problem

$$\mathbf{w}^*, t^* = \arg\min_{\mathbf{w},t} \frac{1}{2}||\mathbf{w}||^2 \qquad \text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i - t) \geq 1, 1 \leq i \leq n$$

On the left hand we have our part of optimisation, on the right hand is our constraint (correctly classify). Using Lagrange multipliers, the dual form of this problem can be derived.

NOTE THIS IS NOT NEEDED TO BE MEMORISED IT IS JUST LEGRANGE FUNCTION
Our problem becomes the following

$$\begin{aligned}\Lambda(\mathbf{w}, t, \alpha_1, \ldots, \alpha_n) &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - t) - 1) \\ &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i y_i (\mathbf{w} \cdot \mathbf{x}_i) + \sum_{i=1}^{n} \alpha_i y_i t + \sum_{i=1}^{n} \alpha_i \\ &= \frac{1}{2}\mathbf{w} \cdot \mathbf{w} - \mathbf{w} \cdot \left( \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \right) + t \left( \sum_{i=1}^{n} \alpha_i y_i \right) + \sum_{i=1}^{n} \alpha_i\end{aligned}$$

(LHS of minus is our minimise, RHS of minus is our constraint) by expansion and rewriting.

taking the partial derivative of our lagrange function with respect to t, and setting it to 0, we find the following.

$$\sum_{i=1}^{n} \alpha_i y_i = 0.$$

Taking the partial derivative of our lagrange function with respect to w, and setting it to 0 we get the following

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

The exact same expression we had for our dual form of perceptron (see notes on dual)! For the perceptron the instance weights are slightly different, they are non negative integers stating how many times an example has been misclassified in training whereas in SVM's they are just non-negative real numbers. They do however have a relationship, that if for a particular example, alpha = 0, that example can be removed from the training set without affecting the learned decision boundary (see figure above, multiplying by 0 cancels out the term we sum). We can further extend this by saying that only for instances where alpha is > 0, do we get support vectors (the training examples nearest to the decision boundary).

This actually allows us to eliminate w and t and give us the following dual lagrangian

$$\begin{aligned}\Lambda(\alpha_1, \ldots, \alpha_n) &= -\frac{1}{2}\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) \cdot \left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) + \sum_{i=1}^{n} \alpha_i \\ &= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i\end{aligned}$$

Since now the problem is finding out values of alpha which make instances > 0. Now any case where alpha is 0, they are ignored in the summation, and what we can also see is the similar to dual perceptron, our gram matrix with the class value included.

## SVM in Dual form

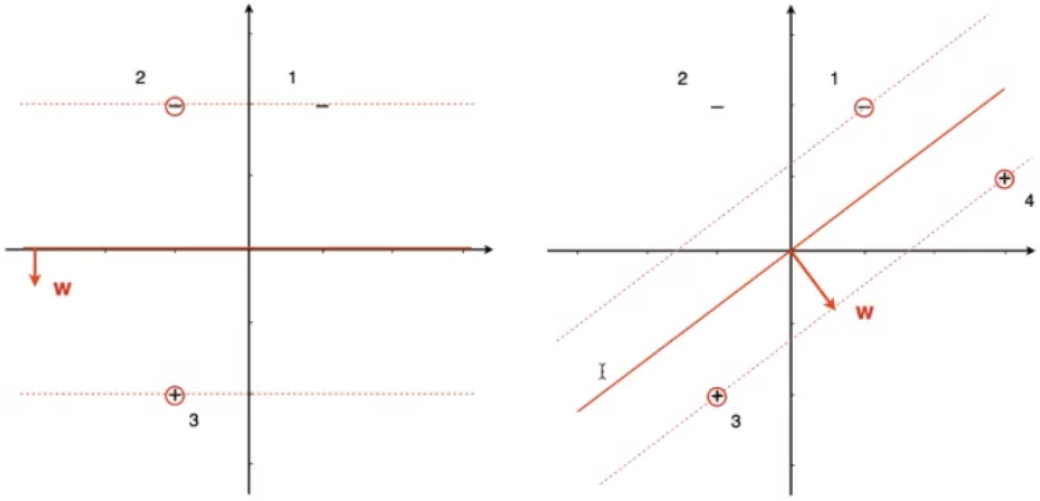
The dual optimisation problem for support vector machines is to maximise the dual lagrangian under positivity constraints and one equality constraint

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1, \ldots, \alpha_n}{\arg\max} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i$$

$$\text{subject to} \quad \alpha_i \geq 0, \quad 1 \leq i \leq n \quad \text{and} \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Notice how this is the inverse of our problem before, now we need to maximise our expression now.

## Example 1




On the left we have a classifier built from three examples with w = (0, -½) and a margin of 2. The circled instances show our support vectors which receive non-zero lagrange multipliers and define our decision boundary. By adding simple another positive example as seen on the right, our decision boundary is rotated to w = (¾, -¼) and the margin is decreased to 1.

## Example 2

Given the following data set, with 3 examples (instances)

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix} \qquad \mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \end{pmatrix}$$

The X' matrix incorporates the class labels into the points (x `dot` y).

We compute the Gram matrix for the case with or without the class label

$$\mathbf{X}\mathbf{X}^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & -5 \\ 3 & 5 & -3 \\ -5 & -3 & 5 \end{pmatrix} \qquad \mathbf{X}'\mathbf{X}'^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & 5 \\ 3 & 5 & 3 \\ 5 & 3 & 5 \end{pmatrix}$$

This gives us the following dual optimization problem.

We need to find the following weights now

$$\arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(5\alpha_1^2 + 3\alpha_1\alpha_2 + 5\alpha_1\alpha_3 + 3\alpha_2\alpha_1 + 5\alpha_2^2 + 3\alpha_2\alpha_3 + 5\alpha_3\alpha_1 + 3\alpha_3\alpha_2 + 5\alpha_3^2\right) + \alpha_1 +$$

$$= \arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1\alpha_3 + 5\alpha_2^2 + 6\alpha_2\alpha_3 + 5\alpha_3^2\right) + \alpha_1 + \alpha_2 + \alpha_3$$

$$\text{subject to } \alpha_1 \geq 0, \alpha_2 \geq 0, \alpha_3 \geq 0 \text{ and } -\alpha_1 - \alpha_2 + \alpha_3 = 0.$$

(note alpha 1 = index 1, alpha 2 = index 2 for our gram matrix, so (0, 0) = alpha 0 squared).
The gram matrix is also this following term from previous part of notes
The condition is that alpha values are larger or equal to 0, and the dot product of the weights with their class sums to 0.

$$\arg\max_{\alpha_1,\ldots,\alpha_n} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

By using our constraint that a3 = a1 + a2 we can substitute to get rid of one of our unknowns, and we get the following

$$\arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(20\alpha_1^2 + 32\alpha_1\alpha_2 + 16\alpha_2^2\right) + 2\alpha_1 + 2\alpha_2$$

By setting partial derivatives to 0 we get the two equations

1. 2 - 20a1 - 16a2 = 0
2. 2 - 16a1 - 16a2 = 0

It is worth nothing that because the objective function is quadratic, the equations for our partial derivatives are guaranteed to be linear

Solving simultaneous equations will give us then

1. A1 = 0
2. A2 = ⅛
3. A3 = ⅛

We then get for our variable w

$$\mathbf{w} = 1/8(\mathbf{x}_3 - \mathbf{x}_2) = \begin{pmatrix} 0 \\ -1/2 \end{pmatrix}$$

Remember w is just the corresponding weight multiplied by the instance index, summed up.
This gives us a margin of 1/||w||, being 2.

From this we can compute the threshold from any support vector, let's use x2 for instance. Since we know that y2 * (w * x2 - t) = 1, we get -1(-1 - t) = 1, therefore t = 0. Note this is taking from the fact that w*x = t +  m, and we set m = 1 for convention.

## Change in model for example 2

Suppose we add a positive instance at (3, 1). This will give us the new following data es:

$$\mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \\ 3 & 1 \end{pmatrix} \qquad \mathbf{X}'\mathbf{X}'^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & 5 & -5 \\ 3 & 5 & 3 & 1 \\ 5 & 3 & 5 & -5 \\ -5 & 1 & -5 & 10 \end{pmatrix}$$

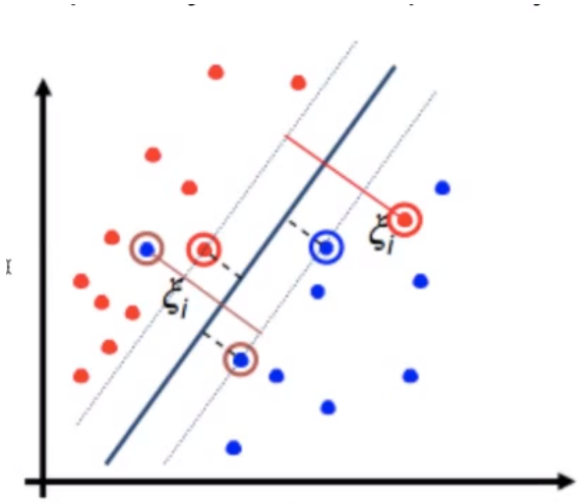
By using the same calculations to the previous, we can show that the margin actually decreases to 1 and the decision boundary w now becomes

$$\mathbf{w} = \begin{pmatrix} 3/5 \\ -4/5 \end{pmatrix}.$$

The Lagrange multipliers now are $\alpha_1 = 1/2$, $\alpha_2 = 0$, $\alpha_3 = 1/10$ and $\alpha_4 = 2/5$. Thus, only $\mathbf{x}_3$ is a support vector in both the original and the extended data set.

## Noise

All work so far assumes linear separability, however misclassified examples can break this assumption.




Note for the maths that is coming up, we only need to know it at a high level, and understand what's going on.
To deal with this issue of noise we introduce slack variables to allow misclassification of instances, this is known as a soft margin as it allows room for error. We now try to minimise the following

$$\Phi(w) = w^T w + C \sum \xi_i$$

Where $\xi_i$ is the slack variable.

Our hyperparameter (user parameter) C will limit how much slackness we allow in the optimisation, it is still a quadratic optimisation problem. Our constraint on C is that it must be larger than our weight, but positive.

C can be gotten from prior knowledge, or in common practice cross-validation.

These slack variables are attached, one for each example, which allow some of them to be inside the margin or even at the wrong side of the decision boundary. These are known as our margin errors. Now we have to change our constraints to the following

$$\mathbf{w} \cdot \mathbf{x}_i - t \geq 1 - \xi_i$$

And we must also add the sum of the slack variables to the objective function as part of our minimisation, giving us the new optimisation problem

$$\mathbf{w}^*, t^*, \xi_i^* = \arg\min_{\mathbf{w},t,\xi_i} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$
$$\text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n$$

These slack variables are a weight of how confident we are that these parameters are on the right side of the decision boundary.

Here we can see that a high value of C means that margin errors will cause high penalty, while low value permits more margin errors (possibly even misclassification) in order to achieve a large margin. If we are confident in our dataset, keeping C low, however if we aren't a high value of C will help.

The more margin errors we allow, the fewer support vectors we need, this is why C controls to an extent the complexity of our SVM and is known as the complexity parameter.

Applying the lagrange function we get the following

$$\Lambda(\mathbf{w}, t, \xi_i, \alpha_i, \beta_i) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(\mathbf{w}\cdot\mathbf{x}_i - t) - (1-\xi_i)) - \sum_{i=1}^{n}\beta_i\xi_i$$
$$= \Lambda(\mathbf{w}, t, \alpha_i) + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i$$

And we can obtain values for our unknowns by obtaining partial derivatives and setting equal to 0

## Performance

It is worth noting that SVM algorithms are massively sped up if the data is spare (many values are 0). This is because they compute a massive amount of dot products (n*n) dot products. It is very efficient to compute dot products with sparse data because we can ignore all zero values. SVMs are able to process spare datasets with tens of thousands of attributes efficiently.

SVMs are great linear classifiers if we don't have too much data (since we dot product a lot) or if we have sparse data.

## Non-Linear SVMS

This is very similar to SVMs, we use the same kernel trick to increase efficiency of our NLSVMs. Overfitting isn't as big of a problem for our hyperplane margin, this is because there are only a few support vectors relative to the size of the training set, and error is based on the number of support vectors regardless of dimensionality.

We compute the dot product before the nonlinear mapping is performed, all computations are done in original low dimensional space.

Instead of computing

$$x = b + \sum_{i \text{ is a support vector}} \alpha_i y_i \mathbf{a}(i) \bullet \mathbf{a}$$

We compute a much more efficient

$$x = b + \sum_{i \text{ is a support vector}} \alpha_i y_i (\mathbf{a}(i) \bullet \mathbf{a})^n$$

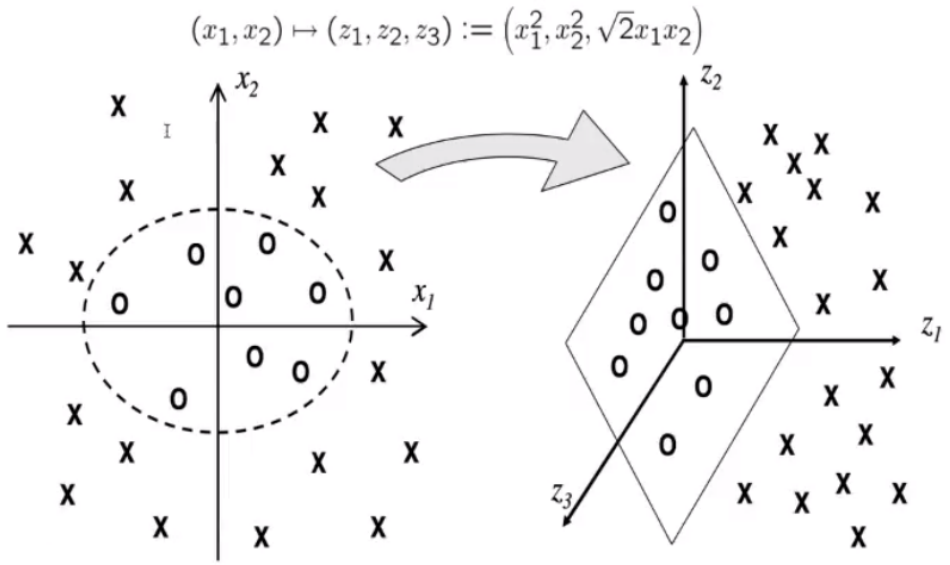
Where n is just the number of factors.

An example of a polynomial function is the following

$$K(x, y) = (1 + x \bullet y)^q$$

## Example

A 2 attribute example where d = 2.




Here we can see a linear classifier simply won't work, to go around this we make a mapping to a 3d feature space..and create a separating hyperplane. This mapping is performed by the kernel function, which computes the dot products of our vectors in a lower dimensional space and raises them to a power n (usually in higher dimension, see example in kernel trick notes earlier).

We can also use other kernel functions instead, it is much more general. A kernel machine in general sense is just a function applied to our dot product before it is handed to the SVM.

$$x = b + \sum_{i \text{ is a support vector}} \alpha_i y_i K(\mathbf{a}(i) \bullet \mathbf{a})$$

Common choices for kernel functions include the following

- polynomial kernel of degree $d$

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \bullet \mathbf{x}_j + 1)^d$$

- radial basis function kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = e\frac{-(\mathbf{x}_i \bullet \mathbf{x}_j + 1)^2}{2\sigma^2}$$

- sigmoid kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\beta \mathbf{x}_i \bullet \mathbf{x}_j + b)$$

## Extensions (Perceptrons)

The perceptron algorithm is a very simple iterative algorithm which tests whether example xi is correctly classified by evaluating the following

$$y_i \sum_{j=1}^{|D|} \alpha_j y_j \mathbf{x}_i \cdot \mathbf{x}_j.$$

- The most important component of this calculation is the dot product of xi and xj
- Assuming bivariate examples, i.e. xi = (xi, yi), xj = (xj, yj) we can write the dot product as the sum of products for each component
    - Xi `dot` Xj = xixj + yiyj
- The corresponding instances in the quadratic feature space are the following

$$\left(x_i^2, y_i^2\right) \text{ and } \left(x_j^2, y_j^2\right), \text{ and their dot product is}$$
$$\left(x_i^2, y_i^2\right) \cdot \left(x_j^2, y_j^2\right) = x_i^2 x_j^2 + y_i^2 y_j^2.$$

- This is almost equivalent to our earlier case where

$$(\mathbf{x}_i \cdot \mathbf{x}_j)^2 = (x_i x_j + y_i y_j)^2 = (x_i x_j)^2 + (y_i y_j)^2 + 2 x_i x_j y_i y_j,$$

    - Note its not the same because the third term of the cross products
- We can capture this term however by extending our feature vector with a 3rd feature (sqrt(2*xy))
- Now when we compute (xi `dot` xj)^2 we will achieve the same result.

This will give us a new feature space in the following

$$\phi(\mathbf{x}_i) = \left(x_i^2, y_i^2, \sqrt{2}x_i y_i\right) \qquad \phi(\mathbf{x}_j) = \left(x_j^2, y_j^2, \sqrt{2}x_j y_j\right)$$
$$\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) = x_i^2 x_j^2 + y_i^2 y_j^2 + 2x_i x_j y_i y_j = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$$

And we can now define our kernel function in the following

$$\boldsymbol{\kappa}(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2,$$

And now we are able to replace xi `dot` xj with the kernel method in the dual perceptron algorithm to obtain the kernel perceptron. This works also for many other kernels that satisfy their conditions.

This now takes our earlier dual perceptron which could only handle linear cases, into a non-linear perceptron due to our feature engineering. This was done by replacing our initial linear dot product, with our kernel that satisfied conditions.

**Algorithm** KernelPerceptron$(D, \eta)$ // perceptron training algorithm using a ke
**Input:** labelled training data $D$ in homogeneous coordinates, plus
kernel function $\boldsymbol{\kappa}$
**Output:** coefficients $\alpha_i$ defining non-linear decision boundary
$\alpha_i \leftarrow 0$ for $1 \leq i \leq |D|$
*converged*←false
**while** *converged* = false **do**
  *converged*←true
  **for** $i = 1$ to $|D|$ **do**
    **if** $y_i \sum_{j=1}^{|D|} \alpha_j y_j \boldsymbol{\kappa}(\mathbf{x}_i, \mathbf{x}_j) \leq 0$ **then**
      $\alpha_i \leftarrow \alpha_i + 1$
      *converged*←false
    **end**
  **end**
**end**

- The decision boundary learned with a non-linear kernel cannot be represented by a simple weight vector in input space. Thus, in order to classify a new example $\mathbf{x}$ we need to evaluate $y_i \sum_{j=1}^{n} \alpha_j y_j \boldsymbol{\kappa}(\mathbf{x}, \mathbf{x}_j)$ which is an $O(n)$ computation involving all training examples, or at least the ones with non-zero multipliers $\alpha_j$.
- This is why support vector machines are a popular choice as a kernel method, since they naturally promote sparsity in the support vectors.
- Although we have restricted attention to numerical features here, kernels can be defined over discrete structures, including trees, graphs, and logical formulae, opening the way to extending geometric models to non-numerical data[1].

## Applications of SVM

- Machine vision
    - Example: Face identification
    - Best accuracy before deep learning
- Handwritten digit recognition
    - Comparable to the best alternative
- Bioinformatics
    - Example: prediction of protein secondary structure
    - Microarray classification
- Text classification
- Algorithm can be modified to deal with numeric prediction problems
    - Support vector regression.

Anywhere with low data samples (since we only need few support vectors) and high dimensionality of data (hopefully sparse) SVM's shine. The kernel methods can be extended into other aspects of machine learning, it is not limited to SVMs.

Optimisation and kernelization are separate, and modular. We can first figure out the optimisation then figure out what kernel to use, they are not tightly coupled.

# Ensemble Learning

## Introduction

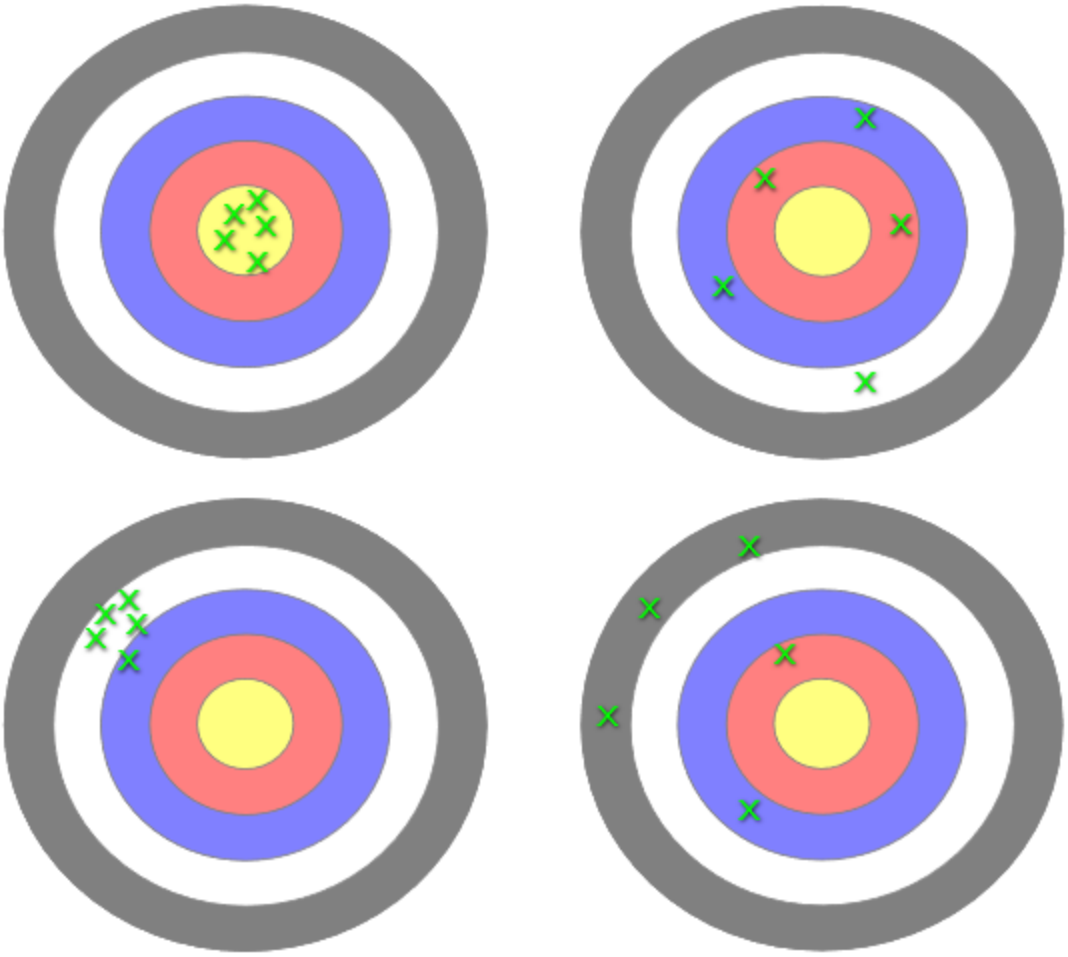
In previous lectures, we introduced some theoretical ideas about limits on machine learning. But do these have any practical impact?
The answer is yes!

- The bias-variance decomposition of error can be a tool for thinking about how to reduce error in learning
- Take a learning algorithm and ask:
    - how can we reduce its bias?
    - how can we reduce its variance?
- Ensemble learning methods can be viewed in this light
- A form of multi-level learning:  learning a number of base-level models from the data, and learning to combine these models as an ensemble

## Review: bias-variance decomposition

- Theoretical tool for analyzing how much specific training set affects performance of classifier
- Assume we have an infinite number of classifiers built from different training sets all of the same size:
    - The bias of a learning scheme is the expected error due to the mismatch between the learner's hypothesis space and the space of target concepts
    - The variance of a learning scheme is the expected error due to differences in the training sets used
    - Total expected error ≈ bias^2 + variance
- Next slide:  a graphical representation of this idea, where distance from target stands for error

# Bias and variance

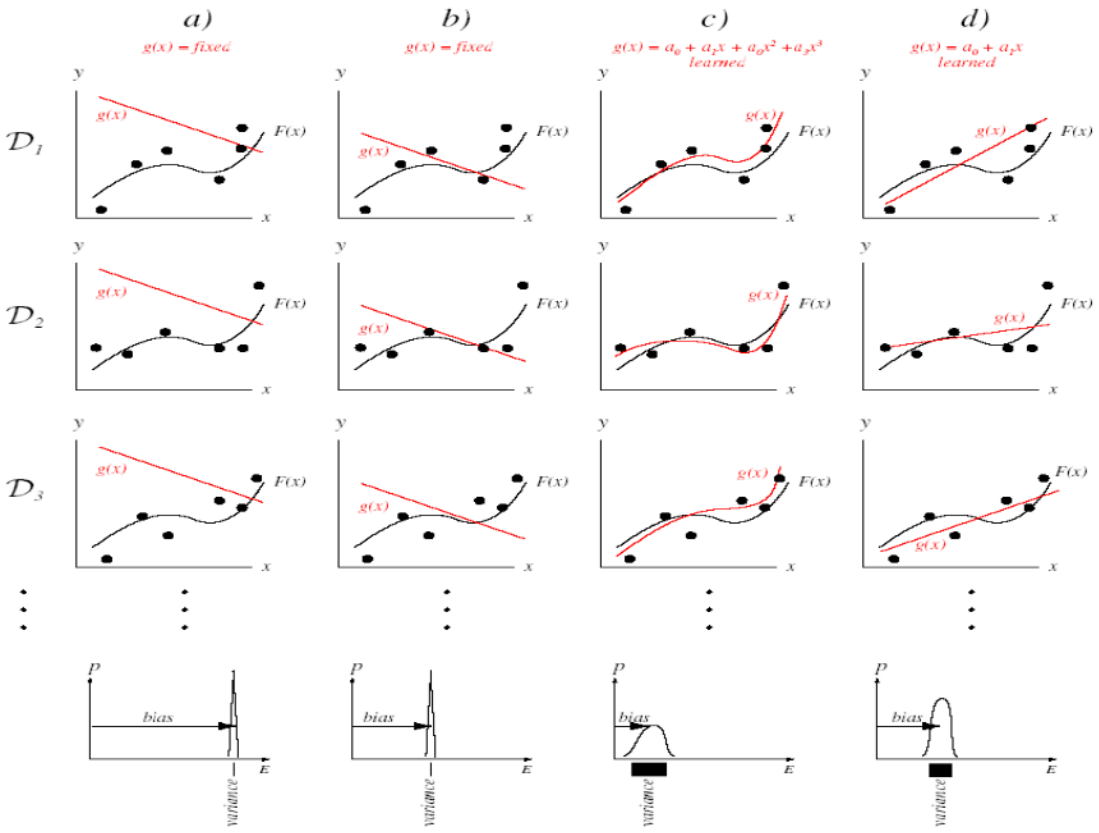


A dartboard metaphor illustrating the concepts of bias and variance. Each dartboard corresponds to a different learning algorithm, and each dart signifies a different training sample. The top row learning algorithms exhibit low bias, on average staying close to the bullseye (the true function value for a particular x), while the ones on the bottom row have high bias. The left column shows low variance and the right column high variance.

## Bias-variance: a trade off

Easier to see with regression in the following figure1 (to see the details you will have to zoom in in your viewer):

- each column represents a different model class g(x) shown in red
- each row represents a different set of n= 6 training points, D_i, randomly sampled from target function F(x) with noise, shown inblack
- probability functions of mean squared error E are shown




- a) is very poor:  a linear model with fixed parameters independent of training data; high bias, zero variance
- b) is better:  a linear model with fixed parameters independent of training data; slightly lower bias, zero variance
- c) is a cubic model with parameters trained by mean-square-error ontraining data; low bias, moderate variance
- d) is a linear model with parameters adjusted to fit each training set;intermediate bias and variance
- training with data n→∞ would give
    - c) with bias approaching small value due to noise
    - but not d)
    - variance for all models would approach zero

## Bias-variance in ensemble classification

- Recall that we derived the bias-variance decomposition for regression– squared-error loss function
- Cannot apply same derivation for classification – zero-one loss
- Bias-variance decomposition used to analyze how much restriction to a single training set affects performance

- Can decompose expected error of any individual ensemble member as follows:
    - Bias = expected error of the ensemble classifier on new data
    - Variance = component of the expected error due to particular training set being used to build classifier
    - Total expected error ≈ bias + variance
- Note (A): we assume noise inherent in the data is part of the bias component as it cannot normally be measured
- Note (B): multiple versions of this decomposition exist for zero-one loss but the basic idea is always the same

## Bias-variance with “Big Data”

- Suppose we have a low bias representation (e.g., all conjunctive concepts),but such concepts may not always occur frequently in small datasets:
- So we can increase bias – e.g., by Naive Bayes-type conditional independence assumptions – but this forces averaging of class distributions over all “small concepts”:

Big Data” may help to resolve the bias-variance dilemma:

- high bias algorithms are often used for efficiency
    - usually simpler to compute
- big data can reduce variance
    - “small” concepts will occur more frequently
    - low bias algorithms can find them in each sample
    - but:  how to compute efficiently ? (Open Problem)

## Bias-variance in “Real-world AI”

Applications increasingly require machine-learning systems to perform at“human-level” (e.g., personal assistants, self-driving vehicles, etc.)

How can an understanding of the bias-variance decomposition help ?

Suppose you are developing an application and you know what“human-level” error would typically be on this task.

You have sufficient data for training and validation datasets, and you are not restricted in terms of the models that you could learn (e.g., from linear regression or classification up to kernel methods, ensembles, deep networks, etc.)

Training-set error is observed to be high compared to human-level – why ?

Bias is too high – solution:  move to a more expressive (lower bias) model

Training-set error is observed to be similar to human-level, but validation set error is high compared to human-level – why ?

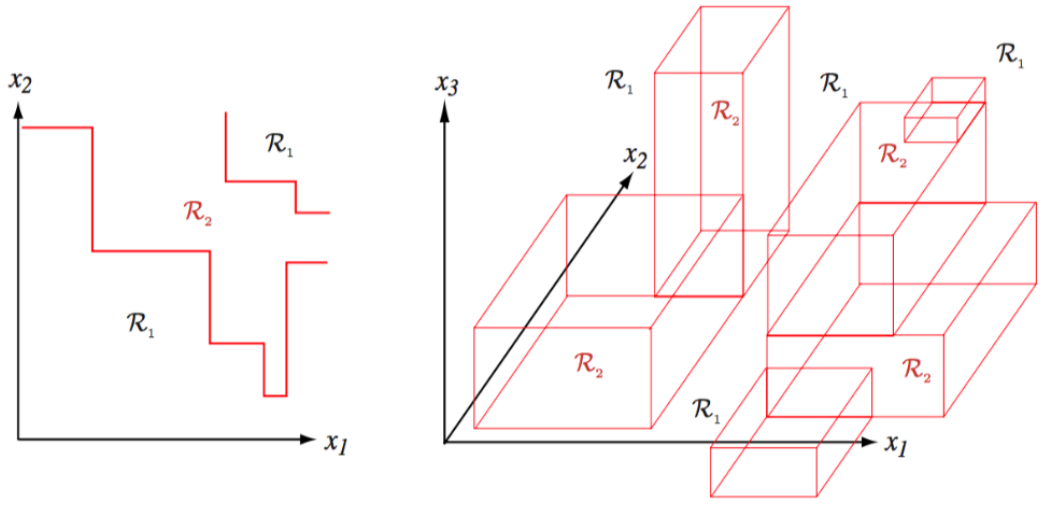
Variance is too high – solution:  get more data (!), try regularization,ensembles, move to a different model architecture

These scenarios are often found in applications of deep learning.

## Stability

- for a given data distribution D
- train algorithm L on training sets S1, S2 sampled from D
- expect that the model from L should be the same (or very similar) on both S1 and S2
- if so, we say that L is a stable learning algorithm
- otherwise it is unstable
- typical stable algorithm: kNN (for some k)
- typical unstable algorithm:  decision-tree learning

## Decision boundaries in tree learning




Decision boundaries for monothetic two-class trees in two and three dimensions; arbitrarily fine decision regions for classes $\mathcal{R}_1$, $\mathcal{R}_2$ can be learned by recursively partioning the instance space.

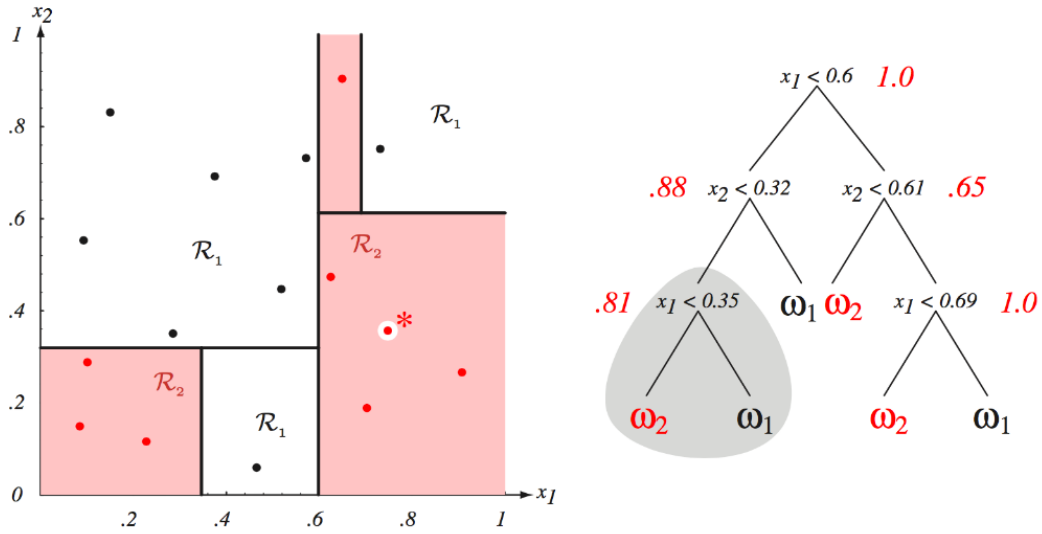
# Instability of Tree Learning

An example shows the effect of a small change in the training data on the structure of an unpruned binary tree learned by CART. The training set has 8 instances for each class:

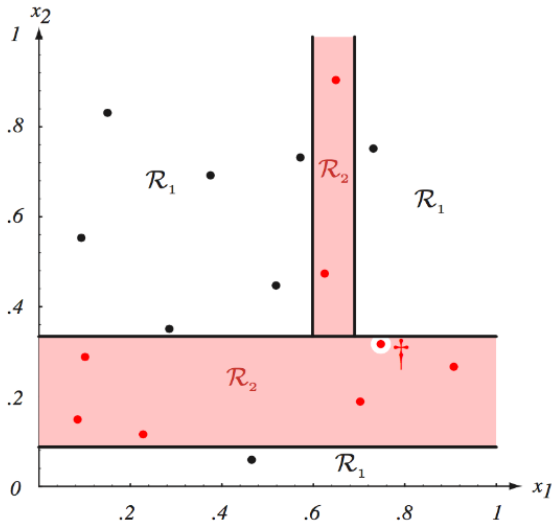
| $\omega_1$ (black) | |
|---|---|
| $x_1$ | $x_2$ |
| .15 | .83 |
| .09 | .55 |
| .29 | .35 |
| .38 | .70 |
| .52 | .48 |
| .57 | .73 |
| .73 | .75 |
| .47 | .06 |

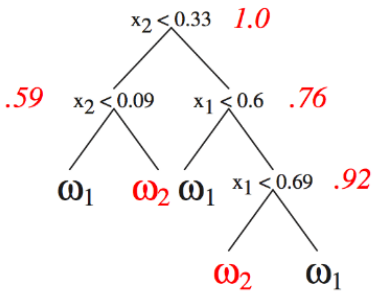
| $\omega_2$ (red) | |
|---|---|
| $x_1$ | $x_2$ |
| .10 | .29 |
| .08 | .15 |
| .23 | .16 |
| .70 | .19 |
| .62 | .47 |
| .91 | .27 |
| .65 | .90 |
| .75 | .36* (.32†) |

Note: for class $\omega_2$ (red) the last instance has two values for feature $x_2$. On the next slide is a tree learned from the data where this instance has value $x_2 = .36$ (marked *), and on the following slide we see the tree obtained when this value is changed to $x_2 = .32$ (marked †).




The partioned instance space (left) contains the instance marked * and corresponds to the decision tree (right).



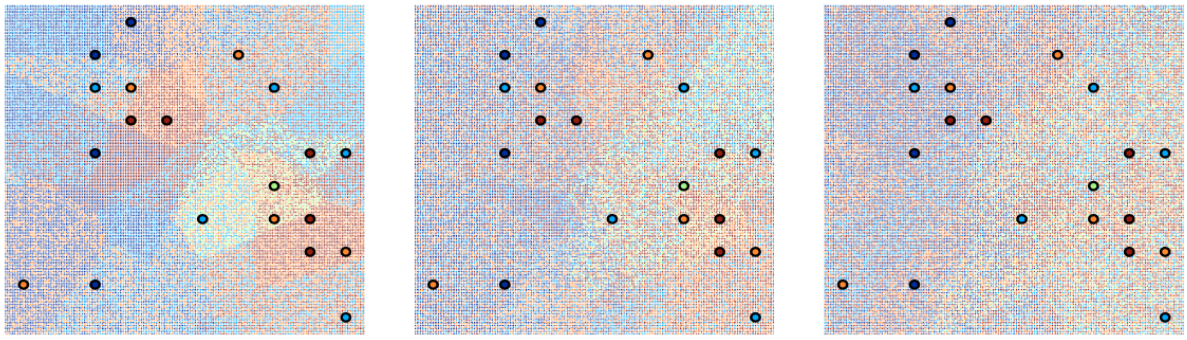




The partioned instance space (left) contains the instance marked † and corresponds to the decision tree (right). Note that both the decision boundaries and the tree topology are considerably changed, for example, testing $x_2$ rather than $x_1$ at the tree root, although the change in data was very small.

## Stability and Bias-Variance

- stable algorithms typically have high bias
- unstable algorithms typically have high variance
- BUT: take care to consider effect of parameters, e.g., in kNN
    - 1NN perfectly separates training data, so low bias but high variance
    - By increasing the number of neighbours k we increase bias and decrease variance (what happens when k=n?)
    - Every test instance will have the same number of neighbours, and the class probability vectors will all be the same !

Decision regions of $k$-nearest neighbour classifiers; the shading represents the predicted probability distribution over the five classes.



3-nearest neighbour | 5-nearest neighbour | 7-nearest neighbour

Illustrates the effect of varying $k$ on stability (i.e., bias and variance).

## Ensemble methods

In essence, ensemble methods in machine learning have the following two things in common:

- they construct multiple, diverse predictive models from adapted versions of the training data (most often reweighted or resampled);
- they combine the predictions of these models in some way, often by simple averaging or voting (possibly weighted).

## Ensembles: combining multiple models

- Basic idea of ensembles or “multi-level” learning schemes: build different “experts” and let them vote
- Advantage: often improves predictive performance
- Disadvantage: produces output that is very hard to interpret
- Notable schemes: bagging, random forests, boosting
    - can be applied to both classification and numeric prediction problems

## Bootstrap error estimation

This is a standard “resampling” technique from statistics. Can be used to estimate a parameter of interest, e.g., error rate of a learning method on a data set.

- sampling from data set with replacement

- e.g.  sample from n instances, with replacement,n times to generate another data set of n instances
- (almost certainly) new data set contains some duplicate instances
- and does not contain others – used as the test set
- chance of not being picked $(1 - 1/n)^n \approx e^{-1} = 0.368$
- 0.632 training set
- error estimate = 0.632×err_test+ 0.368×err_train
- repeat and average with different bootstrap samples
- however, in ML, cross-validation is preferred for error estimates

Why this is interesting/useful
- Can be used to estimate many parameters of interest
- For example, bias, variance, etc.
- Can then apply significance tests and other statistical machinery
- But can be computationally demanding
- Not widely used in machine learning (unlike cross-validation)
- See Bradley Efron's book for more details

"Bootstrap Aggregation"
- Employs simplest way of combining predictions:  voting/averaging
- Each model receives equal weight
- Generalized version of bagging:
    - Sample several training sets of sizen(instead of just having one training set of size n)
    - Build a classifier for each training set
    - Combine the classifiers' predictions
- This improves performance in almost all cases if learning scheme is unstable (i.e. decision trees)

# Bagging

- Bagging reduces variance by voting/averaging, thus reducing the overall expected error, even though datasets are all dependent
    - In the case of classification there are pathological situations where the overall error might increase
    - Usually, the more classifiers the better, with diminishing returns
- Problem:  we only have one dataset!
- Solution:  generate new datasets of size n by sampling with replacement from original dataset, giving duplicate instances
- Can be applied to numeric prediction and classification
- Can help a lot if data is noisy

## Bagging in a nutshell

**Learning** (model generation)
Let n be the number of instances in the training data.
For each of t iterations:
    Sample n instances with replacement from training set.
    Apply the learning algorithm to the sample.
    Store the resulting model.

Classification
For each of the t models:
    Predict class of instance using model.
Return “ensemble” class.

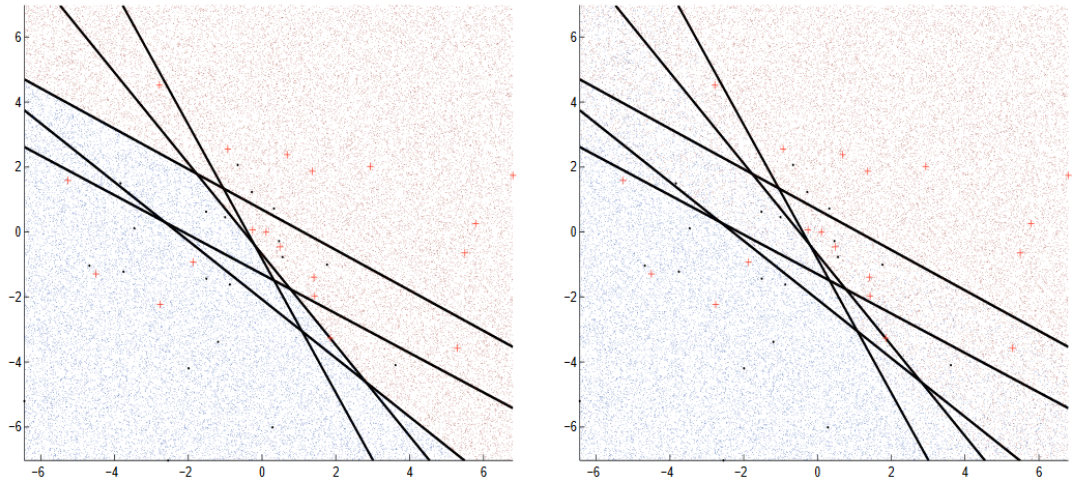
What is the ensemble class ?
The class that has been predicted most often (for classification, i.e., the majority vote or mode), or the mean of the output class values (for regression).

## Bagging more precisely

**Algorithm** Bagging$(D, T, \mathcal{A})$ // train ensemble from bootstrap samples
**Input:** dataset $D$; ensemble size $T$; learning algorithm $\mathcal{A}$.
**Output:** set of models; predictions to be combined by voting or averaging.

1 **for** $t = 1$ to $T$ **do**
2 | bootstrap sample $D_t$ from $D$ by sampling $|D|$ examples with replacement
3 | run $\mathcal{A}$ on $D_t$ to produce a model $M_t$
4 **end**
5 **return** $\{M_t | 1 \leq t \leq T\}$

## Bagging linear classifiers



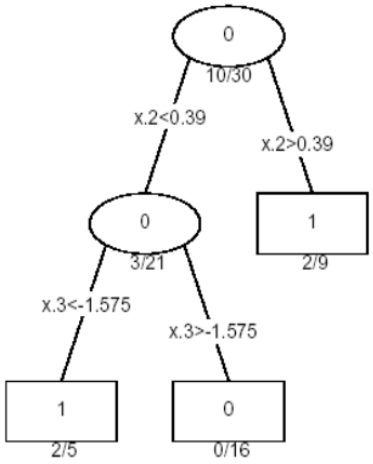
(left) An ensemble of five *basic linear classifiers* built from bootstrap samples with bagging. The decision rule is majority vote, leading to a piecewise linear decision boundary. (right) If we turn the votes into probabilities, we see the ensemble is effectively grouping instances in different ways, with each segment obtaining a slightly different probability.

## Bagging trees

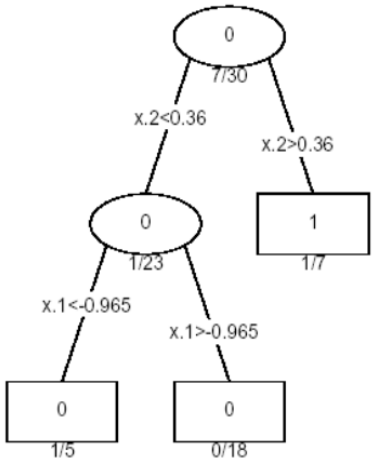
An experiment with simulated data:

- sample of size n= 30, two classes, five features
- Pr(Y= 1 | x_1 ≤ 0.5) = 0.2 and Pr(Y= 1 | x_1 > 0.5) = 0.8)
- test sample of size 2000 from same population
- fit classification trees to training sample, 200 bootstrap samples
- trees are different (tree induction is unstable)
- therefore have high variance
- averaging reduces variance and leaves bias unchanged
- (graph: test error for original and bagged trees, with green – vote; purple – average probabilities)

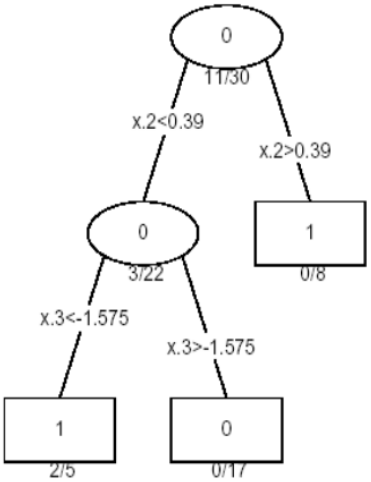
## Original Tree



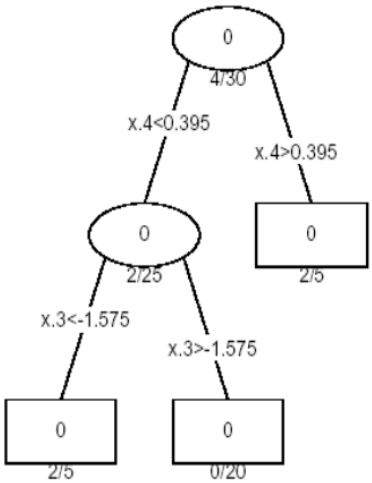

## Bootstrap Tree 1

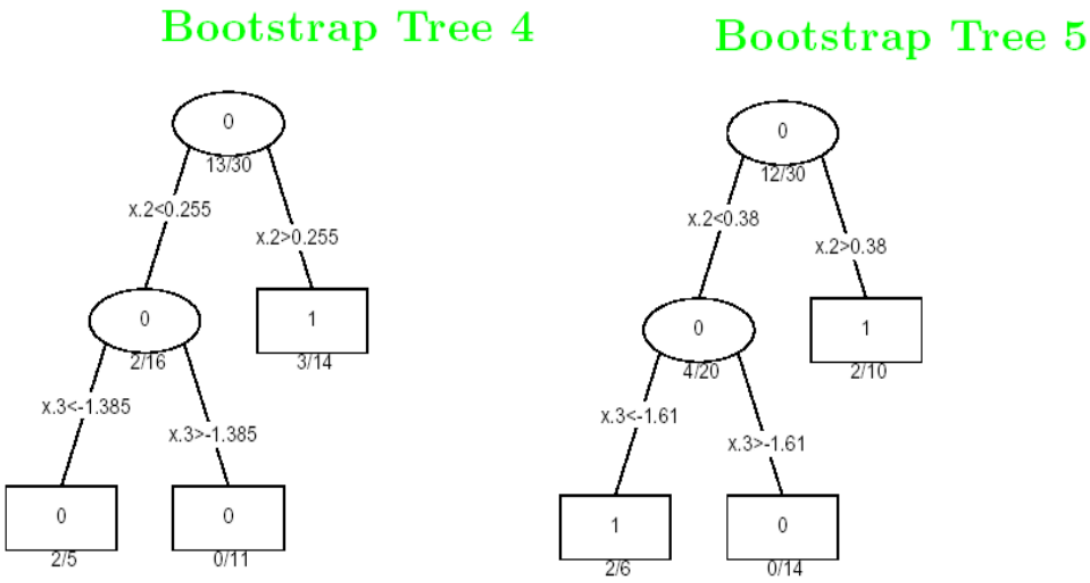


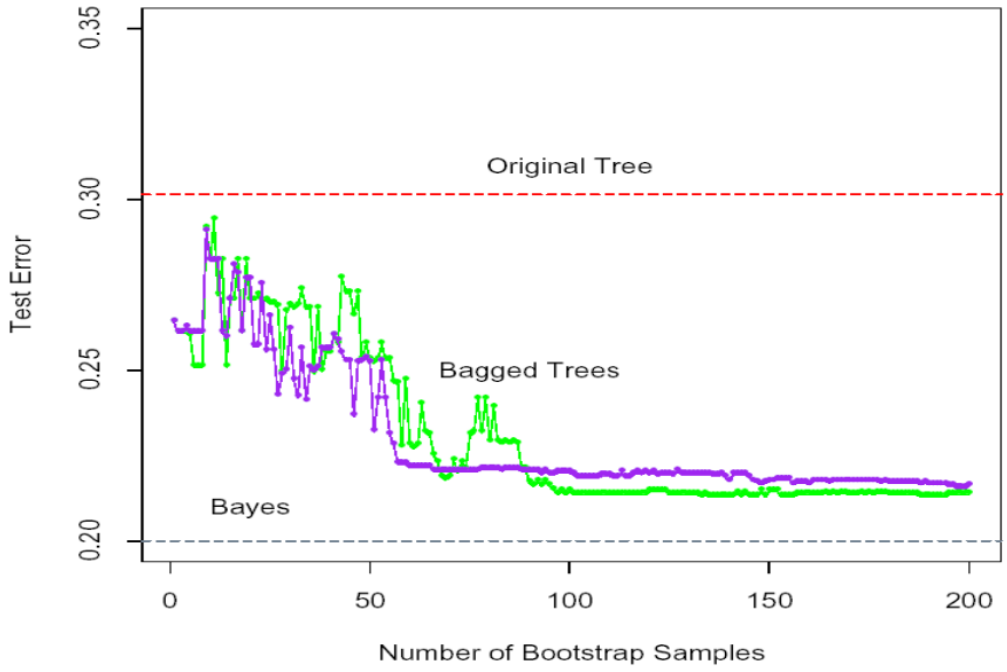

## Bootstrap Tree 2




## Bootstrap Tree 3






The news is not all good:

- when we bag a model, any simple structure is lost
- this is because a bagged tree is no longer a tree…
- …but a forest
- although bagged trees can be mapped back to a single tree…
- ...this reduces claim to comprehensibility

- stable models like nearest neighbour not very affected by bagging
- unstable models like trees most affected by bagging
- usually, their design for interpretability (bias) leads to instability
- more recently,random forests(see Breiman's web-site)

# Random Forests

## Randomization

- Can randomize learning algorithm instead of input to introduce diversity into an ensemble
- Some algorithms already have a random component: e.g., initial weights in a neural net
- Most algorithms can be randomized, e.g., greedy algorithms:
  - Pick N options at random from the full set of options, then choose the best of those N choices
  - E.g.: attribute selection in decision trees
- More generally applicable than bagging: e.g., we can use random subsets of features in a nearest-neighbor classifier
  - Bagging does not work with stable classifiers such as nearest neighbour classifiers
- Can be combined with bagging
  - When learning decision trees, this yields the Random Forest method for building ensemble classifiers

## Random Forests

**Algorithm** RandomForest$(D, T, d)$ // train ensemble of randomized trees
**Input:** data set $D$; ensemble size $T$; subspace dimension $d$.
**Output:** set of models; predictions to be combined by voting or averaging.

**for** $t = 1$ to $T$ **do**
    bootstrap sample $D_t$ from $D$ by sampling $|D|$ examples with replacement
    select $d$ features at random and reduce dimensionality of $D_t$ accordingly
    train a tree model $M_t$ on $D_t$ without pruning
**end**
**return** $\{M_t | 1 \leq t \leq T\}$

Leo Breiman's Random Forests algorithm is essentially like Bagging for trees, except the ensemble of tree models is trained from bootstrap samples and random subspaces.

- each tree in the forest is learned from
    - a bootstrap sample, i.e., sample from the training set with replacement
    - a subspace sample, i.e., randomly sample a subset of features
- advantage:  forces more diversity among trees in ensemble
- advantage:  less time to train since only consider a subset of features

Note:  combining linear classifiers in an ensemble gives a piecewise linear(i.e., non-linear) model, whereas multiple trees can be combined into a single tree.

# Boosting

- Also uses voting/averaging but each model is weighted according to their performance
- Iterative procedure:  new models are influenced by performance of previously built ones
    - New model is encouraged to become “expert” for instances classified incorrectly by earlier models
    - Intuitive justification:  models should be experts that complement each other
- There are several variants of this algorithm . . .

## The strength of weak learnability

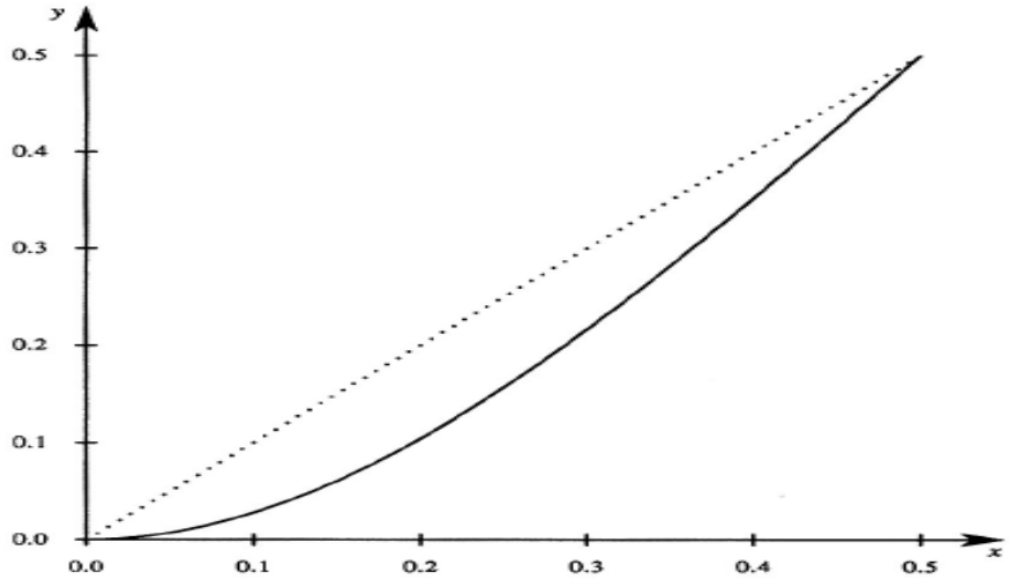
- Learner produces a binary [−1,+1] classifier h with error rate < 0.5.
- In some sense h is “useful”, i.e., better than random !
- Strong learner if < 0.5and “close” to zero.
- Weak learner if < 0.5 and “close” to 0.5.
- Question (arising from Valiant’s PAC framework):
    - Is there a procedure to convert a weak learner into a strong learner ?

Schapire (1990) - first boosting algorithm.
Method:

- weak learner learns initial hypothesis h1 from N examples
- next learns hypothesis h2 from new set of N examples, half of which are misclassified by h1
- then learns hypothesis h3 from N examples for which h1 and h2 disagree
- “boosted” hypothesis h gives voted prediction on instance x:
    - If h1(x) = h2(x) then return agreed prediction, else
    - return h3(x)

Result:  if h1 has error rate <0.5 then error of h bounded by 32−23, i.e., better than(see next slide).
Schapire showed that weak learners can be boosted into strong learners.

## Boosting a weak learner reduces error




A graph of the function $g(x) = 3x^2 - 2x^3$.

## Why does boosting work?

Simple boosting using three classifiers in an ensemble:

- H1 is a weak learner
- dataset used to train h2 is maximally informative wrt h1
- H3 learns on what h1 and h2 disagree about
- for prediction on instance x:
    - If h1 and h2 agree, use that label (probably correct)
    - otherwise use h3 (probably neither h1 or h2 are correct)

Can apply this reasoning recursively within each component classifier

## A general boosting method

- original version:  after initial hypothesis, each subsequent hypothesis has to “focus” on errors made by previous hypotheses
- general version:  extend from 3 hypotheses to many
- how to focus current hypotheses on errors of previous hypotheses ?
- applyweightsto misclassified examples

- called adaptive boosting

## Weight updates in boosting

- Suppose a linear classifier achieves performance as in the first contingency table. The error rate is= (9 + 16)/100 = 0.25.
- We want to give half the weight to the misclassified examples. The following weight updates achieve this: a factor 1/2= 2 for them is classified examples and 1/2(1−) = ⅔ for the correctly classified examples.

| | *Predicted* ⊕ | *Predicted* ⊖ | |
|---|---|---|---|
| *Actual* ⊕ | **24** | **16** | 40 |
| *Actual* ⊖ | **9** | **51** | 60 |
| | 33 | 67 | 100 |

- Taking these updated weights into account leads to the contingency table below, which has a (weighted) error rate of0.5.

| | ⊕ | ⊖ | |
|---|---|---|---|
| ⊕ | **16** | **32** | 48 |
| ⊖ | **18** | **34** | 52 |
| | 34 | 66 | 100 |

## Boosting

**Algorithm** Boosting$(D, T, \mathcal{A})$ // train binary classifier ensemble, reweighting datasets

**Input:** data set $D$; ensemble size $T$; learning algorithm $\mathcal{A}$

**Output:** weighted ensemble of models

$w_{1i} \leftarrow 1/|D|$ for all $x_i \in D$

**for** $t = 1$ to $T$ **do**

  run $\mathcal{A}$ on $D$ with weights $w_{ti}$ to produce a model $M_t$

  calculate weighted error $\epsilon_t$

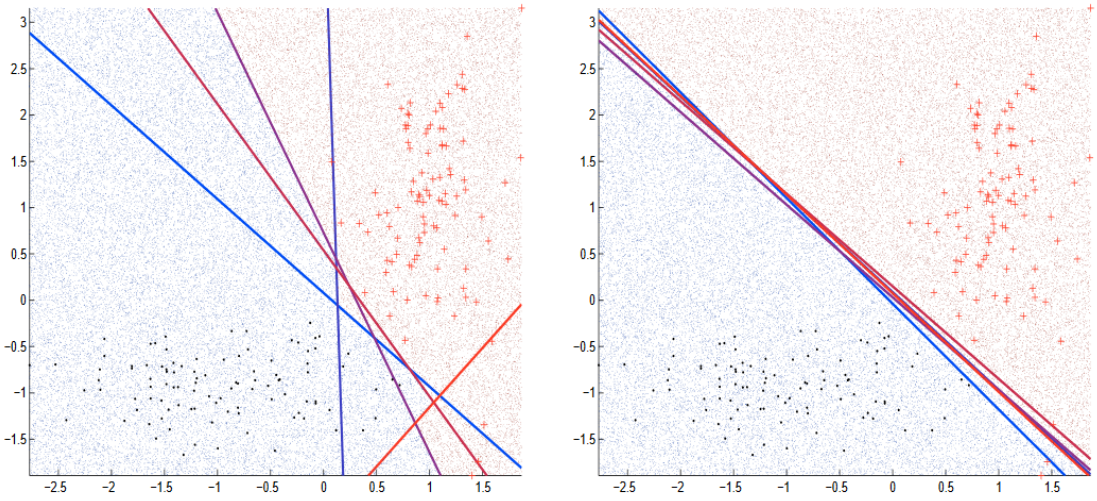
  $\alpha_t \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$

  $w_{(t+1)i} \leftarrow \frac{w_{ti}}{2\epsilon_t}$ for misclassified instances $x_i \in D$

  $w_{(t+1)j} \leftarrow \frac{w_{tj}}{2(1-\epsilon_t)}$ for correctly classified instances $x_j \in D$

**end**

**return** $M(x) = \sum_{t=1}^{T} \alpha_t M_t(x)$



(left) An ensemble of five boosted *basic linear classifiers* with majority vote. The linear classifiers were learned from blue to red; none of them achieves zero training error, but the ensemble does. (right) Applying bagging results in a much more homogeneous ensemble, indicating that there is little diversity in the bootstrap samples.

## Why those αt?

The two weight updates for the misclassified instances and the correctly classified instances can be written as reciprocal terms $\delta_t$ and $1/\delta_t$ normalised by some term $Z_t$:

$$\frac{1}{2\epsilon_t} = \frac{\delta_t}{Z_t} \qquad \frac{1}{2(1-\epsilon_t)} = \frac{1/\delta_t}{Z_t}$$

From this we can derive

$$Z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)} \qquad \delta_t = \sqrt{\frac{1-\epsilon_t}{\epsilon_t}} = \exp(\alpha_t)$$

So the weight update for misclassified instances is $\exp(\alpha_t)/Z_t$ and for correctly classified instances $\exp(-\alpha_t)/Z_t$. Using the fact that $y_i M_t(x_i) = +1$ for instances correctly classified by model $M_t$ and $-1$ otherwise, we can write the weight update as

$$w_{(t+1)i} = w_{ti}\frac{\exp(-\alpha_t y_i M_t(x_i))}{Z_t}$$

which is the expression commonly found in the literature.

## More on boosting

- Can be applied without weights using resampling with probability determined by weights
  - Disadvantage: not all instances are used
  - Advantage: resampling can be repeated if error exceeds 0.5
- Stems from computational learning theory
- Theoretical result: training error decreases exponentially
- Also: works if base classifiers not too complex and their error doesn't become too large too quickly

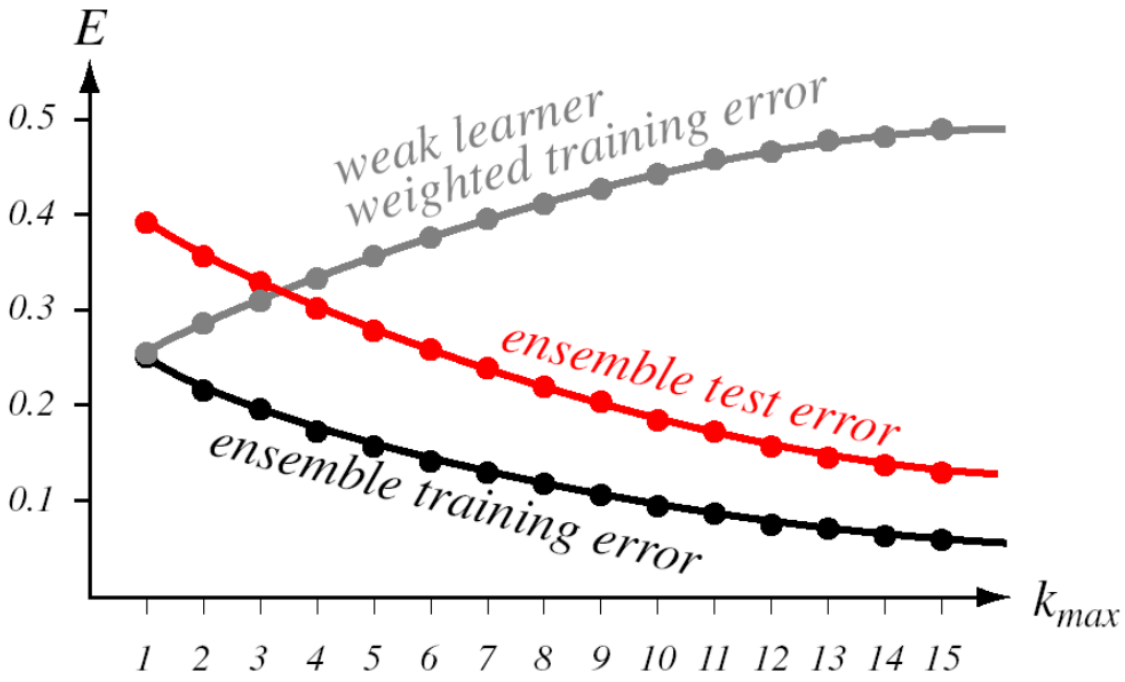
- Puzzling fact: generalization error can decrease long after training error has reached zero
  - Seems to contradict Occam's Razor !
  - However, problem disappears if margin(confidence) is considered instead of error
    - Margin: difference between estimated probability for true class and most likely other class (between -1, 1)
- Boosting works with weak learners: only condition is that error doesn't exceed 0.5 (slightly better than random guessing)

- LogitBoost: more sophisticated boosting scheme in Weka (based on additive logistic regression)

## Boosting reduces error

Adaboost applied to a weak learning system can reduce the training error exponentially as the number of component classifiers is increased.

- focuses on “difficult” patterns
- training error of successive classifier on its own weighted training set is generally larger than predecessor
- training error of ensemble will decrease
- typically, test error of ensemble will decrease also\

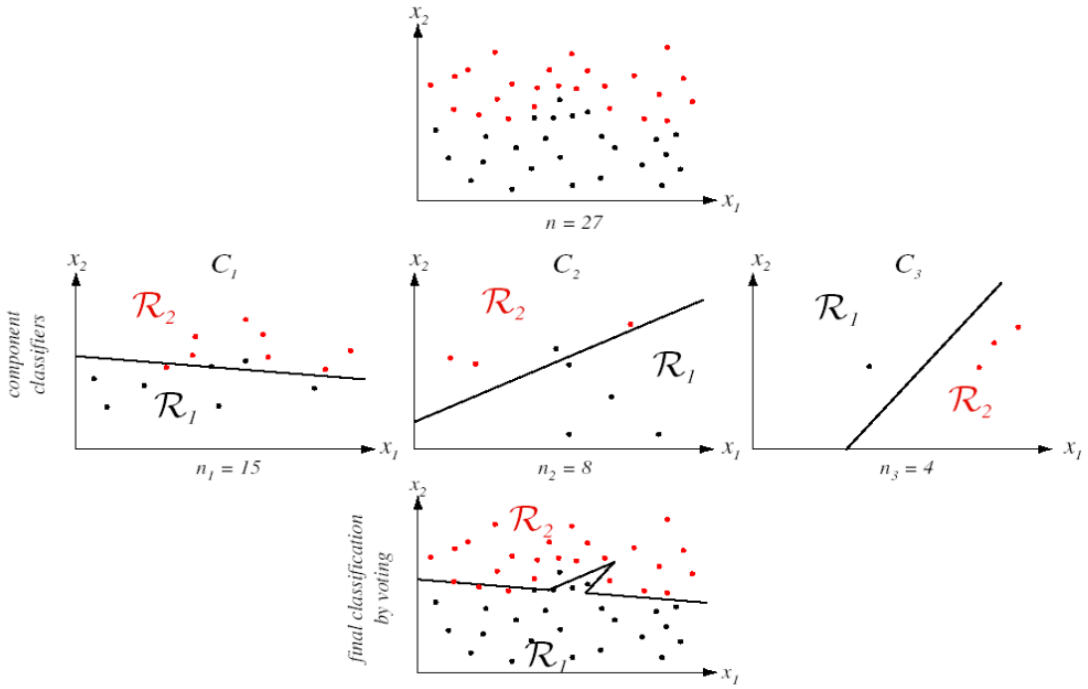



## Boosting enlarges the model class

A two-dimensional two-category classification task

- three component linear classifiers
- final classification is by voting component classifiers
- gives a non-linear decision boundary
- each component is a weak learner (slightly better than 0.5)

- ensemble classifier has error lower than any single component
- ensemble classifier has error lower than single classifier on complete training set




# Other ensemble methods

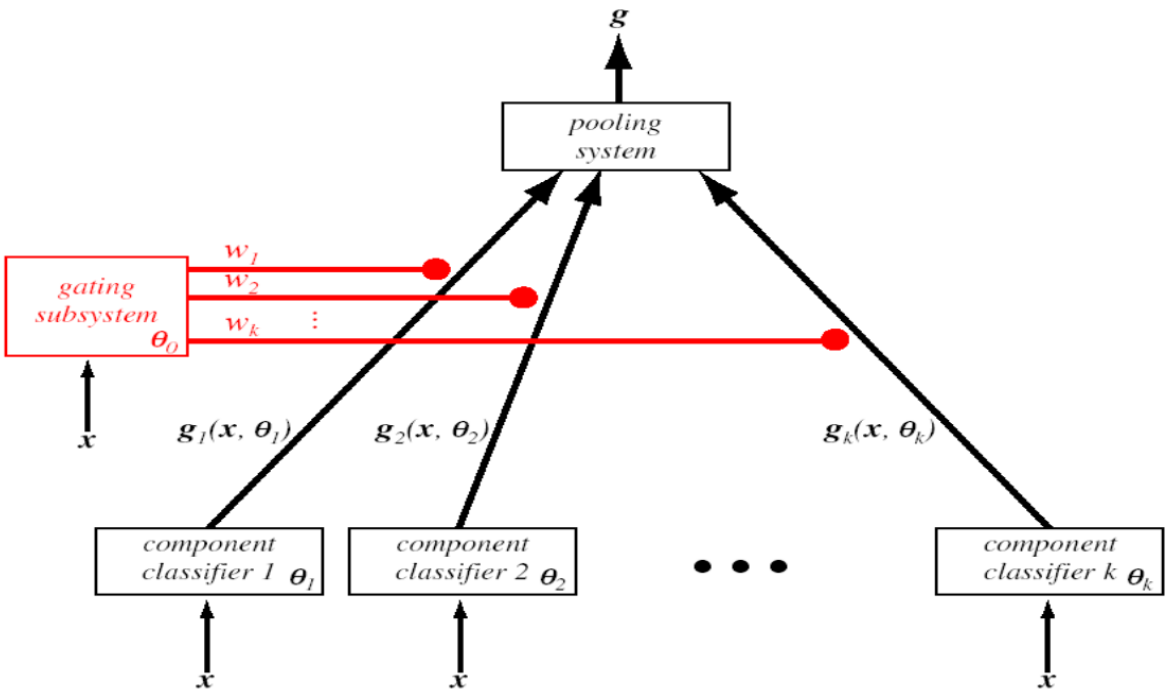
## Stacking

- So far, ensembles where base learners all use same algorithm
- But what if we want to combine outputs of different algorithms ?
- Also, what if the combining method could be tuned from data ?
- Stacked generalization" or stacking
- Uses meta learner instead of voting to combine predictions of base learners
    - Predictions of base learners (level-0 models) are used as input for metalearner (level-1 model)
- Each base learners considered a feature, with value its outputˆyoninstance x
- But predictions on training data can't be used to generate data for the level-1 model!
    - So a cross-validation-like scheme is employed
- If base learners can output probabilities it's better to use those as input to meta learner
    - gives more information to meta-learner
- Which algorithm to use to generate meta learner?
    - In principle, any learning scheme can be applied, but suggested to use
    - relatively global, smooth" models (David Wolpert)

    - Since base learners do most of the work
    - And this reduces risk of overfitting
- Stacking can also be applied to numeric prediction (and density estimation)

## Mixture of Experts

- Framework for learning assuming data generated by a mixture model
    - base level component classifiers (or rankers, . . . )
    - outputs are combined by a tunable system to do the ‘mixing”
- Each component models an “expert” for some part of the problem
- All component outputs are pooled for ensemble output
- Can be trained by gradient descent



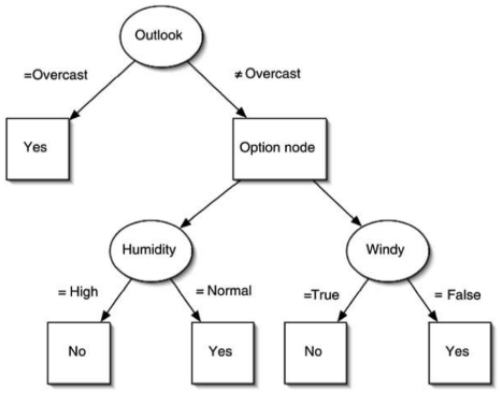

## Additive Regression

- Using statistical terminology, boosting is a greedy algorithm for fitting an additive model
- More specifically, it implements forward stagewise additive modeling
- Forward stagewise additive modeling for numeric prediction:
    - Build standard regression model (e.g., regression tree)
    - Gather residuals, learn model predicting residuals (e.g. another regression tree), and repeat
- To predict, simply sum up individual predictions from all regression models

Gradient (Tree) Boosting is based on this approach, where at each boosting iteration a model is fit to approximate the components of the negative gradient of the overall loss Friedman (2001).

## Option Trees

- Ensembles are not easily interpretable
- Can we generate a single model?
    - One possibility: ”cloning” the ensemble by using large amounts ofartificial data that is labeled by the ensemble
    - Another possibility: generating a single structure that represents anensemble in a compact fashion
- Option tree: decision tree with option nodes
    - Idea: follow all possible branches at option node
    - Predictions from different branches are merged using voting or byaveraging probability estimates

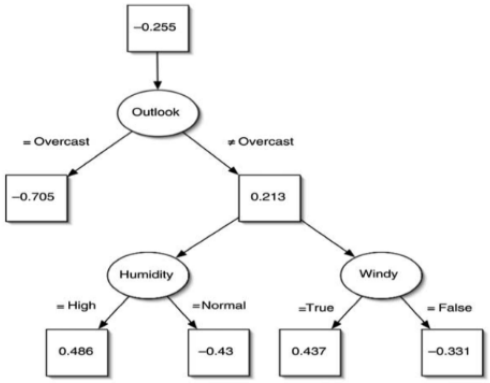



- Can be learned by modifying a standard decision tree learner:
    - Create option node if there are several equally promising splits (within a user-specified interval)
    - When pruning, error at option node is average error of options

## Alternating Decision Trees

- Can also grow an option tree by incrementally adding nodes to itusing a boosting algorithm
- The resulting structure is called an alternating decision tree, with splitter nodes and prediction nodes
    - Prediction nodes are leaf nodes if no splitter nodes have been added tothem yet

- Standard alternating tree applies to 2-class problems but the algorithm can be extended to multi-class problems
- To obtain a prediction from an alternating tree, filter the instance down all applicable branches and sum the predictions
- Predictions from all relevant predictions nodes need to be used,whether those nodes are leaves or not
- Predict one class or the other depending on whether the sum is positiveor negative




- Different approaches, but can be grown using a boosting algorithm:
  - Assume that the base learner used for boosting produces a single conjunctive if-then rule in each boosting iteration, including numeric prediction
  - Choose best extension among all possible extensions applicable to the tree, according to the loss function used

## Gradient Boosting

- Boosting algorithms learn a form of additive model
  - training uses a forward stagewise procedure
- At each boosting iteration, a new (weighted) component function is added to the boosted model
- In gradient boosting, this approach is used to solve an optimization problem
  - Informally, need to minimize loss over all components (basis functions)over all training examples
- A simpler approximation to this optimization is a forward stepwise procedure
  - At each iteration, minimize loss summed over all previously added components, plus the current one

    - In gradient boosting, a regression tree is learned at each iteration to minimize the loss for predicting the negative gradient at each leaf
- Implemented in the widely-used XGBoost package for scalable learning

# Unsupervised Learning

As datasets get larger and larger, it becomes difficult to get a good set of labels.

Supervised learning is when we known what classes, and we find the definition of our model in terms of the data, this includes problems such as

- Classification
- Discriminant analysis
- Class prediction
- Supervised pattern recognition

In unsupervised learning, we don't know the class and need to discover the class with the definitions from the data. This includes problems such as

- Cluster analysis
- Class discovery
- Unsupervised pattern recognition

In methods such as clustering, we address the problem of assigning instances to classes given only by observations about the instances. Without giving the labels we try to find what they are from the cluster.

In practice, any feature can be our label, the difficulty comes with finding this label. However, oftentimes the class is not a known feature, in some cases our class is not measured.

Unsupervised learning is good for

- Simplifying a problem (dimensionality reduction)
- Exploratory data analysis (visualization)
- Data transformation to simplify a classification problem
- Group data instances into subsets
- structuring
- Learning new features (later use in classification)
- Track data change (concept drift) over time
- To learn generative models for images, text, video, speech etc.

## Dimensionality Reduction

Each numeric feature in a dataset is a dimension. (n features → n dimensional space). For all algorithms, we don't really have a restriction on the number of dimensions, and oftentimes we don't need all the features. Many features can be related (opposite of Naive Bayes). In most

cases keeping all the features in our dataset won't really improve the models and in some cases even worsen the models. This can also be a problem for feature selection since if we remove features that are related, we can return arbitrary features without their related feature.

To solve this issue we can find a set of new features that are smaller than the original set, and combine the related features to do this reduction.

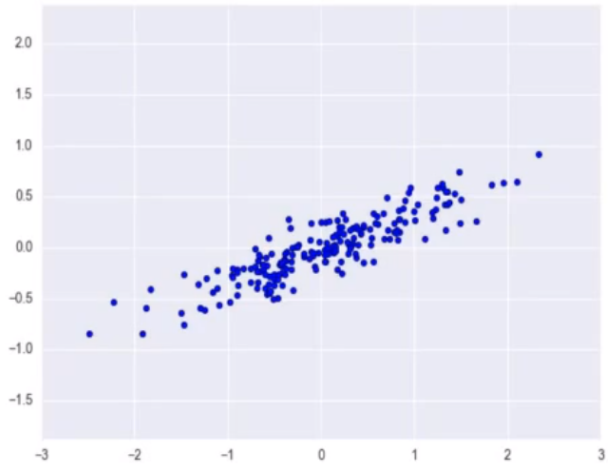
- Smaller than original set
- Preserves information (in the original set)

## PCA (Principal Component Analysis)

We want to look for features in a transformed space such that each dimension in the new space captures the variation in the original data when projected in the dimension

Any new features should be highly correlated with some of the original features but not with any of the new features. One approach can be to use the variance-covariance matrix that we learnt from correlation and regression.

Example




Here PCA looks for linear combinations of our original features. Using this data we want to transform and reformulate our data. (here there is no distinguishing dimension for our class). In previous examples y axis represented class, but here x and y are both data.

By running PCA, we find two new features on which the original data can be projected. Here we find  two new features which are linear combinations of the initial features, e.g. if we had x1, x2, we could get a new feature like kx1+yx2. These new features are then projected, rotated and scaled trying to maximise the variance. (see figure below) note that the new features are orthogonal to each other.

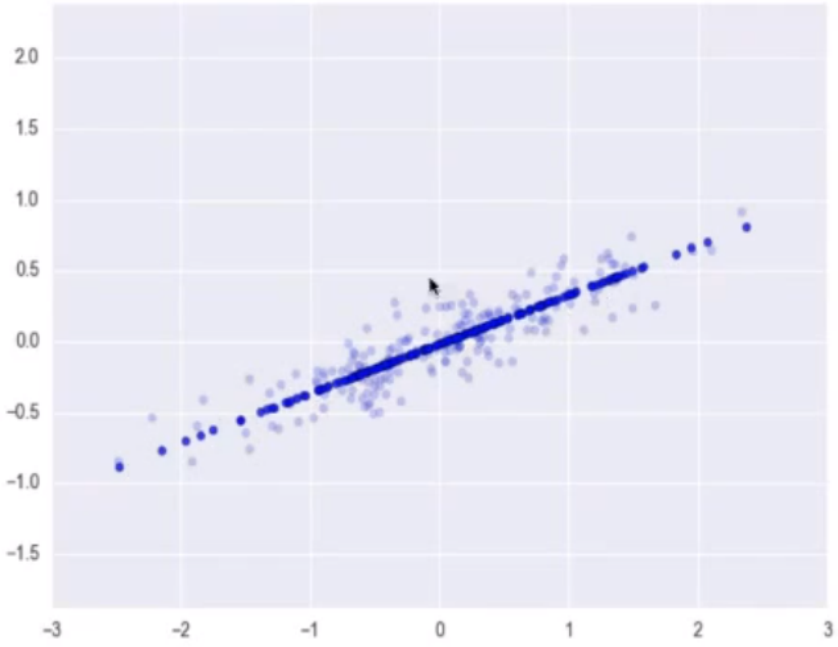
## PCA Algorithm

This algorithm can be presented in several ways, the following shows a basic way in terms of variance reduction

1. Take data as a n * m matrix X
2. Centre the data by subtracting the mean of each column
    a. For each column find the mean
    b. Subtract the values to get mean for the column
3. Construct the covariance matrix C from our centred matrix.
4. Compute eigenvector matrix V (rotation) and eigenvalue matrix S (scaling) such that
    a. $\mathbf{V}^{-1}\mathbf{C}\mathbf{V} = \mathbf{S}$, and $\mathbf{S}$ is a diagonal $m \times m$ matrix
    b. m is the number of features (remember n = number of examples m = number of features)
    c. We have transformed our data into a new set of features by rotation and scaling.
5. Sort columns of S in decreasing order (decreasing variance).
6. Remove columns of S below a hyperparameter threshold.

Note that this is just feature transformation not feature reduction, to get feature reduction we need to select from this matrix a subset of features smaller than m.

By applying this algorithm to our above example and rejecting the second component, we half the dimensionality while maintaining a lot of the original variance, this is seen by plotting the inverse transform of this component with the original data.



Our aim here is just to take the original data and transform it whilst maintaining the distance relations to the original data.

PCA is typically computed using the Singular Value Decomposition (SVD). The complexity however is cubic based on the number of original features since we perform matrix inversion and computation, so oftentimes it is not feasible for datasets with large dimensionality. This leads us to alternative approaches to compute PCA. this includes

- Random projections which as the name suggests takes random projections
    - This is more scalable but we lose quality
    - However it shows that it preserves the distance relations from the original data.

## Latent Semantic Analysis

Our goal is to find topics in text (groupings).

We have a set of documents (d) and a set of terms (d) and using this we construct a (t x d) matrix X which contains.

- Rows corresponding to the terms
- Columns containing the number of occurrences of a term in a document

Our LSA decomposes our matrix using SVD as the following

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

S contains now singular values sorted in decreasing order. Now we restrict S to have k topics, and for some value k, the restricted decomposition is the following

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$$

$$\begin{bmatrix} & \tilde{X}_{\mathrm{I}} & \end{bmatrix}_{t\times d} = \begin{bmatrix} & U_k & \end{bmatrix}_{t\times k} \begin{bmatrix} S_1 & & \\ & \ddots & \\ & & S_k \end{bmatrix}_{k\times k} \begin{bmatrix} & V_k^T & \end{bmatrix}_{k\times d}$$

This will find us new features which are linear combinations of the original features, in the new k space. This approximation is optimal in a least squares sense, meaning that the original contains just X the word count in the document, and in our new transformed space, we contain

new values however they have the same distances between them, we preserve the relation between the data. Can be seen as the same data, different numbers.

Latent Semantic Analysis (LSA) factorizes a word count matrix for $t$ terms from $d$ documents using SVD to find a number $k < d$ of topics. Here $\mathbf{U}$ and $\mathbf{V}$ have orthogonal columns, and the diagonal matrix $\mathbf{S}$ has the topic "strength" sorted in decreasing order. Restricting $k$ effectively reduces the dimensionality of the document matrix. The matrix $\mathbf{U}_k$ captures the mix of words in each topic. Topics are combined in appropriate proportions by the matrix $\mathbf{V}_k^T$ to "generate" each document.

# Recommendation

Recommender systems are typically data based on large and sparse matrices containing ratings. In our matrix, every item is rated (or interacted with) by a user. Ratings or our interactions are normally on a numeric scale, and this then gives us the following matrix

- M users x N items

Element (x, y) contains the rating y, for user x.
Since most values are missing (not everyone leaves reviews or interacts) we get a large and sparse matrix, this can be problematic, so we apply matrix decomposition.

## Example

Consider this following matrix with 4 items rated by 6 users

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix}$$

Now if these 4 items were rated, we can see that the most popular item is the last one, since its column average is 1.5, and the first item is the least popular with a column average of 0.5. We need to find a way to get structure from this matrix.

Applying our PCA algorithm and LSA, we get the following decomposition

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here we can see that matrix 1 shows people's preference in genre. Matrix 3 shows the genre each item relates to (e.g. crime/comedy etc.). Matrix 2 shows the users scoring (weighting), higher number = more preference, for the following items in terms capturing the users genre preference. Here we created a new feature, genre,,and found a correlation between genre and items, this allowed us to break down our feature space from the 4 items, into 3 genres (shrinking feature space).

Deep learning can be used for this recommendation, and give more personalised recommendations.

# Dimensionality Reduction Summary

- PCA will transform original features to new feature space
- Every new feature is a linear combination of original features, unrelated to other new features
- We aim for new dimensions and maximise variance
- Order by decreasing variance and remove those below a threshold
- Algorithm (PCA) applies matrix operations to translate, rotate and scale original matrix
- Based on covariance matrix which can be kernelised by KernelPCA.
- There are more alternatives to SVD including
    - Random projections
    - Independent component analysis
    - Multidimensional scaling
    - Word2vec
    - Many many more.

# Clustering

- Finding groups of items that are similar
- Unsupervised method
- Class of any data instance is not known
- Success of clustering is often measured subjectively
    - Main means for EDA (exploratory data analysis)
    - Problematic if we need quantitative results
    - Primarily we use it for visualisation and statistical analysis

The dataset for clustering is identical to a dataset for classification, however we just don't have the class.

Clustering algorithms have two main approaches

1. Hierarchical methods
2. Partitioning methods

Hierarchical algorithms are either agglomerative (bottom-up) or divisive (top-down). In most cases, agglomerative methods are used due to the efficient exact algorithms available, they also allow for the user a dendrogram or tree which can be visualised.

Partitioning methods are less widely used due to not being able to have a visual output, however they are extremely useful because they can give us quantitative output of where our class lies in structure and where it should be.

## Representation

Let $N = \{e_1, \ldots, e_n\}$ be a set of elements, i.e. instances.

Let $\mathcal{C} = (C_1, \ldots, C_l)$ be a *partition* of $N$ into subsets.

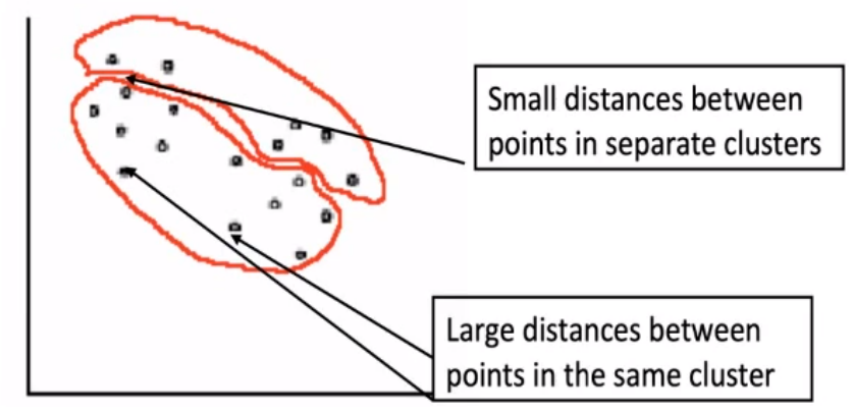
Each element of C can be seen as a cluster, and C is our clustering (model).
Our input data has two forms

1. The original matrix of data, each feature measurement
2. (Feature vectors) Pairwise similarity data between elements pre-computed e.g.
    a. Distance between each element
    b. Correlation

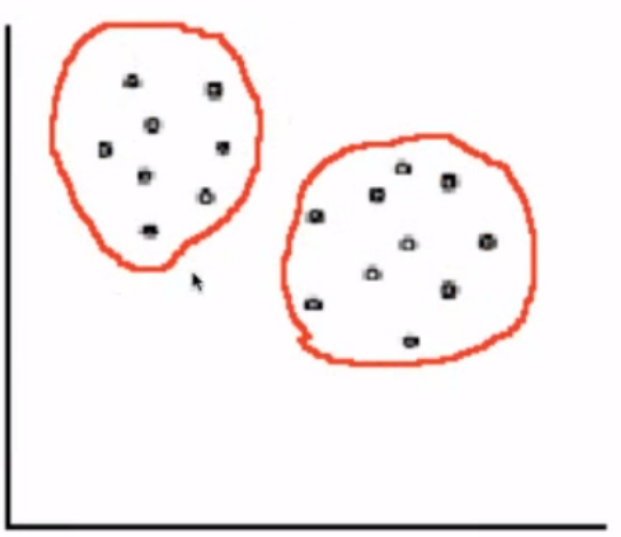
Feature vectors have more information, but similarity is generic since we removed the original feature data.

## Clustering algorithm

Goal of clustering is to find a partition of N instances into homogenous well-separated clusters. Elements in the same cluster should have high similarity, while elements from different clusters should have low similarity.in all cases, we need to define what it means for homogeneity and separation. In practice we use a distance measure appropriate to the problem to perform clustering.
Here is an example of a bad clustering, and good clustering respectively




Bad clustering



Good clustering

This lets us see how to optimise our clustering.

## K-means clustering

1. Set value for k (number of clusters) either by prior knowledge or by search
2. We choose points for centres of our k-clusters at random and apply the following algorithm
    a. Assign each instance to the closest of our centres (whichever it's closest to)
    b. Reassign k-points to be the means of each of the k-clusters.(now we set the mean to be the centroid of our actual data)
    c. Repeat a and b till we converge to reasonably stable clustering.
3. Inductive bias of k-means is that there is a function or set of functions that are picked, one of them generates the data, and the algorithms reconstruct the parameters on these functions.
    a. K means is based on the assumption that the data is normally distributed around the cluster centres, or in a general sense a user can define this distribution. This enforces a certain shape to our clusters which may or may not apply (e.g. circular clusters in 2d space or spheres in 3d space)

P(i) is the cluster assigned to element i, and c(j) is the centroid of the cluster j. d(v1, v2) is the euclidean distance between feature vectors v1 and v2.

Now our algorithm goal is to find a partition P such that the error (distance function)

$$E_P = \sum_{i=1}^{n} d(i, c(P(i))$$

Is at a minimum, where n is the number of data points.
The quality of our solution depends on these factors

- The distribution of the points in the dataset
- The choice of k
- The choice of the location to initialize the centroids.

K-means algorithm

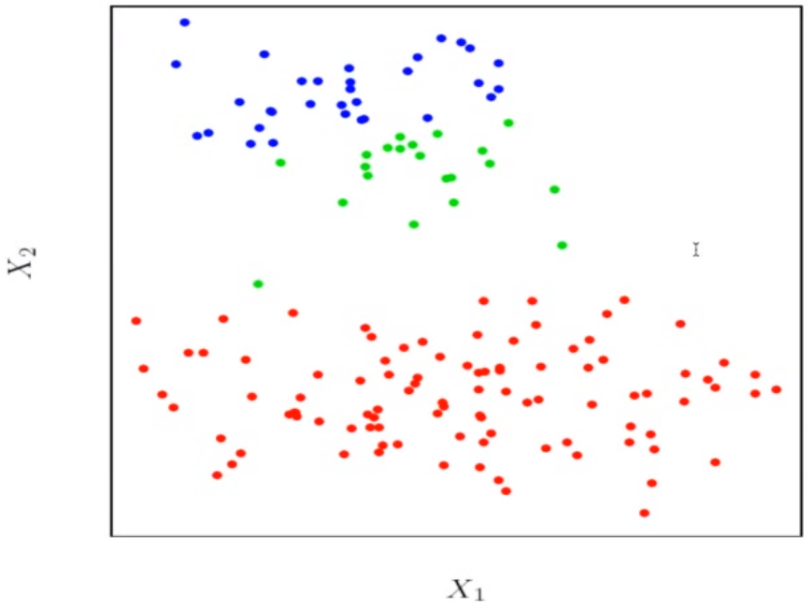
/* feature-vector matrix $M(ij)$ is given */

1. Start with an arbitrary partition $P$ of $N$ into $k$ clusters
2. for each element $i$ and cluster $j \neq P(i)$ let $E_P^{ij}$ be the cost of a solution in which $i$ is moved to $j$:
    1. if $E_P^{i^*j^*} = min_{ij} E_P^{ij} < E_P$ then move $i^*$ to cluster $j^*$ and repeat step 2 else halt.

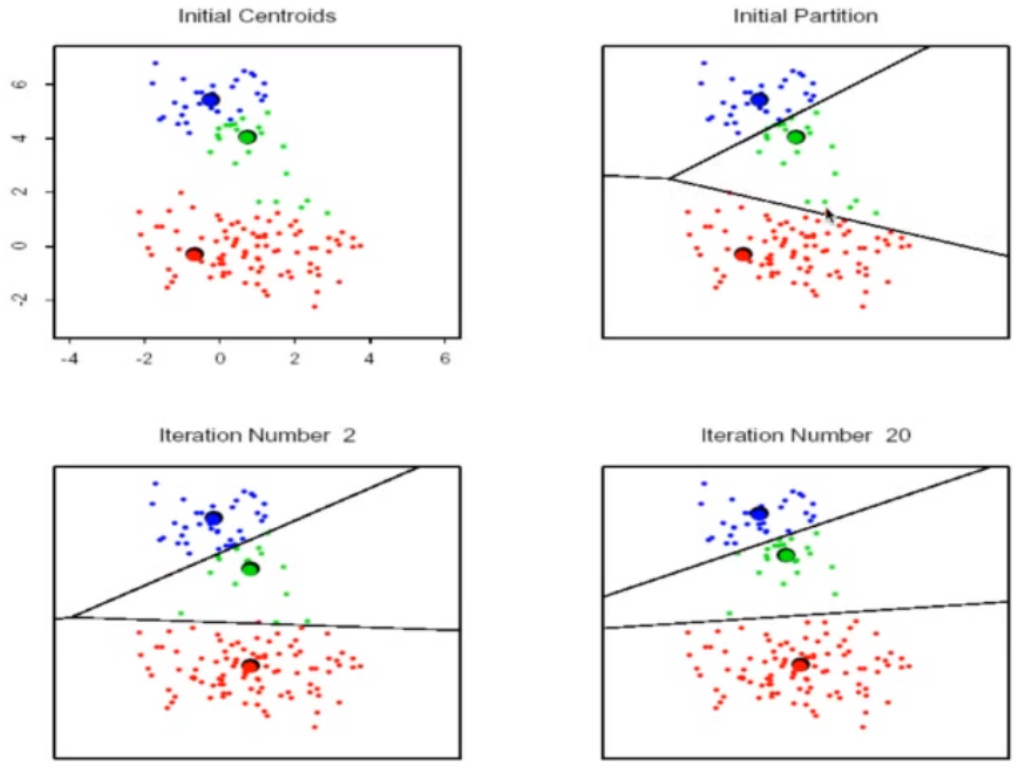
For each element, we find which cluster they are not in, and compute the distance to move the element into that cluster. We want to then move the element into the cluster if it reduces the error. We essentially move the elements into different clusters if it reduces the overall cost.

## Issues

The number of clusters k has a great impact on our solution, picking a wrong k will give us a wrong convergence which is not well separated, for example if for the figure below we use k = 3 we will get a bad separation




In here 3 clusterings would give us a bad model since we can see the red points make a cluster, and the green and blue points can make a second cluster, however if we specify k means with 3 clusters, we will create the following




Here our algorithm can't fix itself and say no, 2 clusters are better, it is stuck with 3.

This is also a greedy algorithm, meaning it is susceptible of being stuck in local minimums, example being

- Four instances at the certicles of a two dimensional rectangle
- Local minimums are the two cluster centers at the midpoint of the rectangle's long side.

The results can vary greatly based on our starting hyperparameters which gives rise to large variance. Similar to how a single tree is not good, we can do the same approach of random forests, we can restart with different random seeds and compare. With reasonably structured data we can see a majority of examples will show the same clustering show us which one is wrong,

# The EM algorithm

This is the generalisation of the k-means algorithm known as expectation maximization.
It should be used when

1. Data is only partially observable (cant see all the data/there is missing data)
2. Unsupervised learning (e.g. clustering)
3. Supervised learning when some instance attributes are unobservable
    a. Missing attributes

Uses for EM algorithm include

- Training bayesian belief networks
- Unsupervised clustering (k-means, AUTOCLASS)
- Learning hidden markov models (Baum-Welch algorithm)

## Finite Mixtures

Each instance x is generated by

1. Choosing one of the k Guassians (normal distributions) with uniform probability.
2. Generating an instance at random according to that normal distribution.

This is called finite mixtures since there is only a finite number of generating distributions being represented.

Given the following

1. Instances from X generated by a mixture of k-gaussian distributions
2. Unknown means of the k-gaussians (assuming all same standard deviation)
3. We don't know which instance was generated by which gaussian

We would like to determine the maximum likelihood estimates of our means.

The algorithm has to learn parameters for it's models, to get EM to do this we allow our model to have a variable/placeholder to check which gaussian generated the variable.

Think of full description of each instance as $y_i = \langle x_i, z_{i1}, z_{i2} \rangle$, where

- $z_{ij}$ is 1 if $x_i$ generated by $j$th Gaussian, otherwise zero

For all these data points there will be an estimated zi for each gaussian, so if we have k = 3, and 100 data points, we would have 3 gaussians and 300 total z points (z1, z2, z3)

- Our xi is observable from the dataset
- Zi is not observable

To estimate these values of z, we will rework k-means into an EM framework

## Implementation

### E step (Expectation)

First we pick a random initial "hypothesis"

$$h = \langle \mu_1, \mu_2 \rangle$$

Before iteration, we need to compute the expected value $E[z_{ij}]$ for each hidden variable z. Assuming our current hypothesis holds (similar to k-means we have our 2 centroids) that the variable z belongs to one of the two clusters, this calculation is the following

$$\begin{aligned} E[z_{ij}] &= \frac{p(x = x_i | \mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)} \\ &= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}} \end{aligned}$$

The numerator calculates the probability that the variable x has the value xi, given the gaussian used was gaussian j. The denominator is the same thing, however for all alternative options. (in this case we only have 2 gaussians)..
Using the probability of a standard gaussian we simplify to the second line.

When computing this probability we notice something very important, in the numerator we notice that:

$$(x_i - \mu_j)^2$$

This is just the distance from that data point to the mean, so the closer we are to the mean of the gaussian, the larger the probability, and vice versa.

This is almost the same as k-means, whereas in k-means we used euclidean distance and tried to minimise distance, here it uses the same principle just with distance to the mean of our gaussians.

Using this expected value, we check if we can make a better estimate for our means.

### M-step (maximisation)

We calculate a new maximum likelihood hypothesis

$$h' = \langle \mu_1', \mu_2' \rangle$$

Using the expected values calculated before to create the new means, this becomes our next iteration hypothesis. The new means are calculated in the following

$$\mu_j \leftarrow \frac{1}{m} \sum_{i=1}^{m} E[z_{ij}] x_i$$

Where m represents the number of instances.

This will give us a local maximum likelihood hypothesis (greedy algorithm). And will provide us with estimates of our hidden variables.

The local maximum is the following

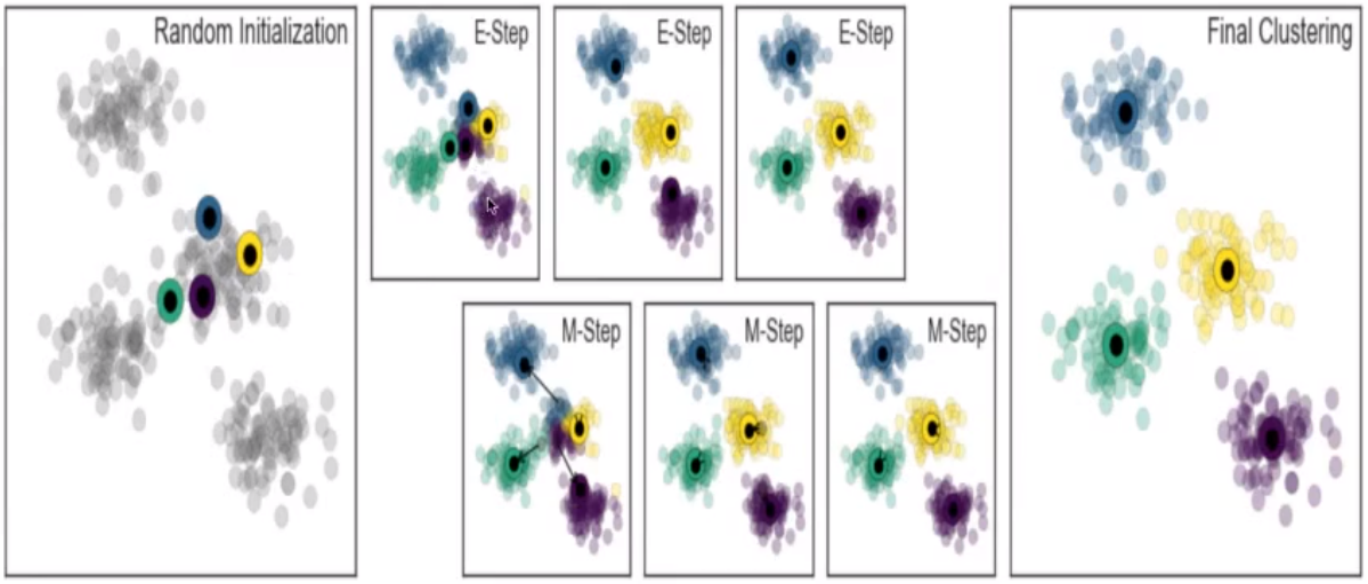
$$E[\ln P(Y|h)]$$

Where Y is our totally unchanged dataset (initial dataset).

## Bringing it together

The E-step (expectation) calculates probabilities for unknown parameters for each instance. The M step (maximisation) will estimate parameters for next iteration based on these probabilities. One way to visualise this is thinking of this as a graph problem. In k-means we have a simple graph where we have nodes as the K-clusters and M data points k `X` m graph. In k-means we simply say there is an edge from the cluster to the datapoint if and only if the instance belongs to the cluster. (checking if edge exist, a graph of true and false will do). With EM however we have an edge from each instance to each cluster and each of these edges contain the weight of probability that it belongs to the cluster (similar to weighted graph).

This diagram outlines each step of the EM algorithm




E-step will calculate strength of association, M-step will assign to a cluster and iterate.

## General EM Method

We define a likelihood function which calculates the new estimates given old estimates from our data set in the following

$$Q(h'|h) \leftarrow E[\ln P(Y|h')|h, X]$$

Where Y is the dataset containing both our data and unknowns

The E step will compute the likelihood function using the current hypothesis and observed data X to estimate the probability distribution over our data set.

$$Q(h'|h) \leftarrow E[\ln P(Y|h')|h, X]$$

The m step will replace our hypothesis h by the hypothesis h' that maximises the Q function.

## Summary of EM

Given:

- Observed data $X = \{x_1, \ldots, x_m\}$
- Unobserved data $Z = \{z_1, \ldots, z_m\}$
- Parameterized probability distribution $P(Y|h)$, where
  - $Y = \{y_1, \ldots, y_m\}$ is the full data $y_i = x_i \cup z_i$
  - $h$ are the parameters

Determine:

- $h$ that (locally) maximizes $E[\ln P(Y|h)]$

## Extending the mixture model

We are able to use more than two distributions, in our case we assumed the same standard deviation. We had also assumed that if we had several attributes, that they were independent features. However this is the easy way out and a naive Bayes assumption which won't always hold. If we have correlated attributes we need to use a bivariate normal distribution with a covariance matrix. If we have n attributes, we would need to estimate n^2 parameters with this approach.

Similarly nominal attributes (discrete) are easy if we assume independence, but when we don't use our naive bayes assumption, it is difficult since two correlated attributes will result in v1 `x` v2 parameters.

The whole purpose of this is that the EM model can be used in many different settings, and what changes is the model.

# Hierarchical clustering

There are two approaches to hierarchical clustering

1. Bottom up approach where at each step we join the two closest clusters starting with single instance clusters
    a. We must consider our decision metric to measure distance between clusters for example
        i. Two closest instances in clusters
        ii. distance between the means
2. Top down approach where we find two clusters and recursively find the two subsets
    a. Very fast
    b. Not widely used in practice

Both these methods will produce a dendrogram (a tree of clusters) which is very easy to read

## Hierarchical agglomerative algorithm

/* dissimilarity matrix $D(ij)$ is given */

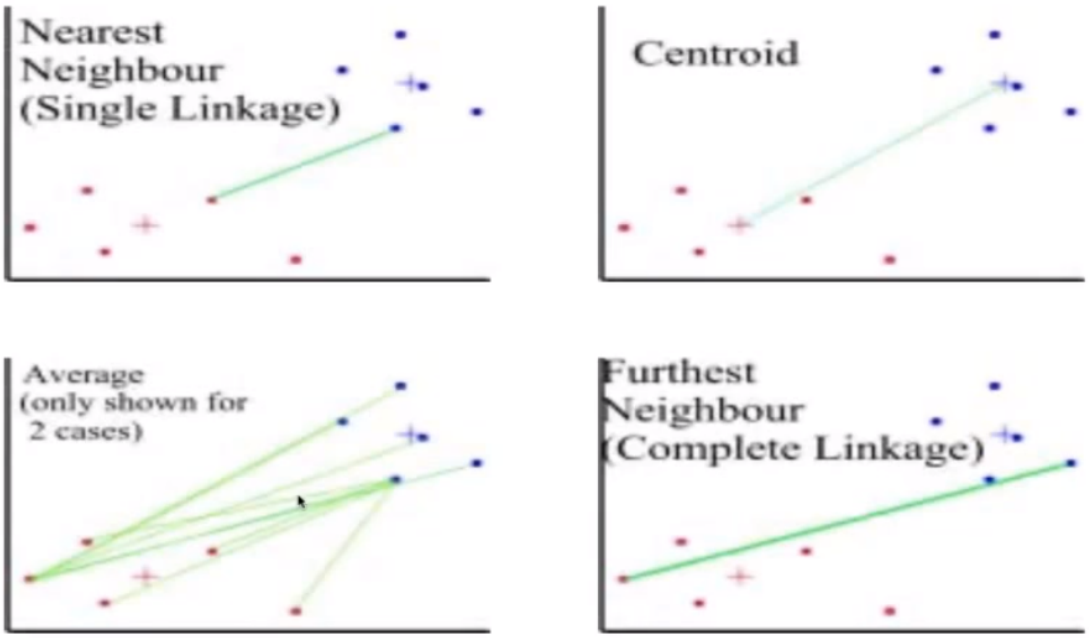
1. Find minimal entry $d_{ij}$ in $D$ and merge clusters $i$ and $j$
2. Update $D$ by deleting column $i$ and row $j$, and adding new row and column $i \cup j$
3. Revise entries using $d_{k,i\cup j} = d_{i\cup j,k} = \alpha_i d_{ki} + \alpha_j d_{kj} + \gamma |d_{ki} - d_{kj}|$
4. If there is more than one cluster then go to step 1.

Find two clusters closest together zombine, and recompute the distance, do this until we have everything in the same cluster

**Single linkage** $d_{k,i\cup j} = \min(d_{ki}, d_{kj})$ and $\alpha_i = \alpha_j = \frac{1}{2}$ and $\gamma = -\frac{1}{2}$.
**Complete linkage** $d_{k,i\cup j} = \max(d_{ki}, d_{kj})$ and $\alpha_i = \alpha_j = \frac{1}{2}$ and $\gamma = \frac{1}{2}$.
**Average linkage** $d_{k,i\cup j} = \frac{n_i d_{ki}}{n_i+n_j} + \frac{n_j d_{kj}}{n_i+n_j}$ and $\alpha_i = \frac{n_i}{n_i+n_j}, \alpha_j = \frac{n_j}{n_i+n_j}$ and $\gamma = 0$.

This algorithm relies on a general updating formula. Having different operations and coefficients gives us different versions of the algorithm causing variant clusterings, examples include

The diagram below shows the different distance measure for each one




### Issues

There are two things we should be aware of when creating dendrograms with our hierarchical structure.

1. Tree structure isn't unique for given clustering, for each bottom up merge, the subtrees must be specified in 2^(n - 1) ways to permute the n leaves in a dendrogram
2. Hierarchical clustering has an inductive bias
    a. The clustering forms a dendrogram despite the possible lack of an implicit hierarchical structure in the data.

It is good practice to run multiple clustering algorithms to be sure we are clustering correctly displayed by the dendrogram.

Changing the structure of linkage we use (distance metric) greatly affects our dendrogram so we also need to be sure of which distance metric to use for our problem.

## Inductive Bias

- K-means clusters spherical seperationa round centroids
- Hierarchical agglomerative clusters form a tree (dendrogram)
    - Data is hierarchical

These inductive bias' may be too strong and not justived and can be relaxed, for example by using kernel k-means we are able to map our features into a different feature space (Similar to how we solved linear separation problems).

# DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is a way to define clusters as dense regions of data instances. Density is based on having more than a minimum number of points with a radius r, and it does not force all points in the dataset to be members of clusters (helps deal with noise).

Core points are points which have more than the minimum points within a radius r. Border points are ones that have less than the minimum points within radius r, but is within r of a core point. All other points are classes as noise.

The DBSCAN algorithm is simple

1. Label all points as either core/border/noise
2. Make a cluster for each point, or set of connected core points
    a. Core points are connected if they are within the radius we set of each other
3. Assign each border point to the cluster of the corresponding core point.

# Cluster quality visualisation

The key idea behind this is to compare each object's separation from other clusters, relative to the homogeneity of its cluster. So for each object we define its silhouette width in the following

$$s(i) \in [-1, 1]:$$

We then let a(i) be the average dissimilarity between i and the elements of the cluster it belongs to.

We also let d(i, C) be the average dissimilarity of the i to elements of some other cluster C.

Finally we let b(i) be the measure that gives us the minimum for d (the cluster which gives us the minimum).

We use this to computer our silhouette width s(i) in the following

$$\frac{b(i) - a(i)}{\max(a(i), b(i))}$$

## How many clusters?

There are many methods for estimating the correct number of clusters. Here we mention two using some clustering criterion.

1. Compare the value for a single cluster to sum of values for breaking cluster into two
    a. If there is a significant reduction then we keep the two clusters.
2. Formalising the elbow detection
    a. Widely implemented

## Clustering Summary

- $k$-means avoids some of these problems, but also has drawbacks
- cannot extract "intermediate features" e.g., a subset of features in which a subset of objects is co-expressed
- for all of these methods, can cluster objects or features, but not both together (coupled two-way clustering)
- should all the points be clustered ? modify algorithms to allow points to be discarded
- visualization is important: dendrograms and alternatives like Self-Organizing Maps (SOMs) are good but further improvements would help; see also T-SNE for visualization
- algorithms like DBSCAN avoid some of these issues
- how can the quality of clustering be estimated ?
    - if clusters known, measure proportion of disagreements to agreements
    - if unknown, measure homogeneity (average similarity between feature vectors in a cluster and the centroid) and separation (weighted average similarity between cluster centroids) with aim of increasing homogeneity and decreasing separation
    - sihouette method, etc.
- clustering is only the first step - mainly exploratory; classification, modelling, hypothesis formation, etc.

# Learning Theory

Learning theory aims at a body of a theory that captures all important aspects of the fundamentals of the learning process and any algorithm or class of algorithms designed to do the learning. We desire the theory to capture the algorithm independent aspects of machine learning.

We want theory in order to relate

- Probability of successful learning
- Number of training examples
- Complexity of hypothesis space
- Runtime complexity of learning algorithm (we want to be efficient)
- Accuracy at which our target concept is approximated
    - What bounds on the error can be obtained
- The way the training examples are presented
    - We assume that all the training data is just loaded into memory in all our cases
    - There is ways for algorithms to go out and get data itself and receive more data not just in 1 batch but inline
        - We can provide examples as we go

We also want to ask ourselves the following in a general sense

- Sample complexity
    - How many training examples do we need for the learner to converge with a high probability to a successful hypothesis
- Computational complexity
    - How much computational effort is required for the learner to converge with a high probability to a successful hypothesis
- Hypothesis complexity
    - How do we measure the complexity of our hypothesis (target function)
    - How large is our hypothesis space
    - Are we overfitting?
- Mistake bounding
    - How many misclassifications of training data will be needed before we converge to a successful hypothesis?

We need to define what it means to be a successful hypothesis

- Is it identical to the target concept?
- Does it mostly agree with the target concept?
- For a majority of runs, will it be robust?
- How reliable is it?

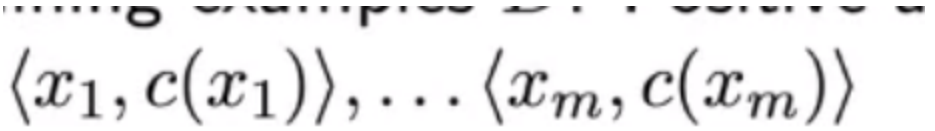
# PAC framework

The probably approximately correct (PAC) framework can be used to address questions such as

- How many training examples
- How much computational effort
- How complex a hypothesis class needed
- How to quantify hypothesis complexity
- How many mistakes will there be

We will look at PAC learning using concept learning (two class classification problems). Given the following

- Instances X with the following attributes
    - Sky
    - Air temperature
    - Humidity
    - Wind
    - Water
    - Forecast
- Target function c
    - enjoysport(x)
        - Gives us a 0 if not enjoy
        - Gives us a 1 if enjoy
- Hypothesis H: a conjunction of literals (feature attribute values)
    - For sky, clear/cloudy
    - For airtemp, cold/hot
    - For humidity, high/low
    - For wind, windy/calm
    - For water, rain/no rain
    - For forecast, good/bad
- Training examples D: set of positive and negative examples of our target function using m samples




    -
- A set of possible target concepts C
- Training instances generated by a fixed, unknown probability distribution D over our set of instances X.

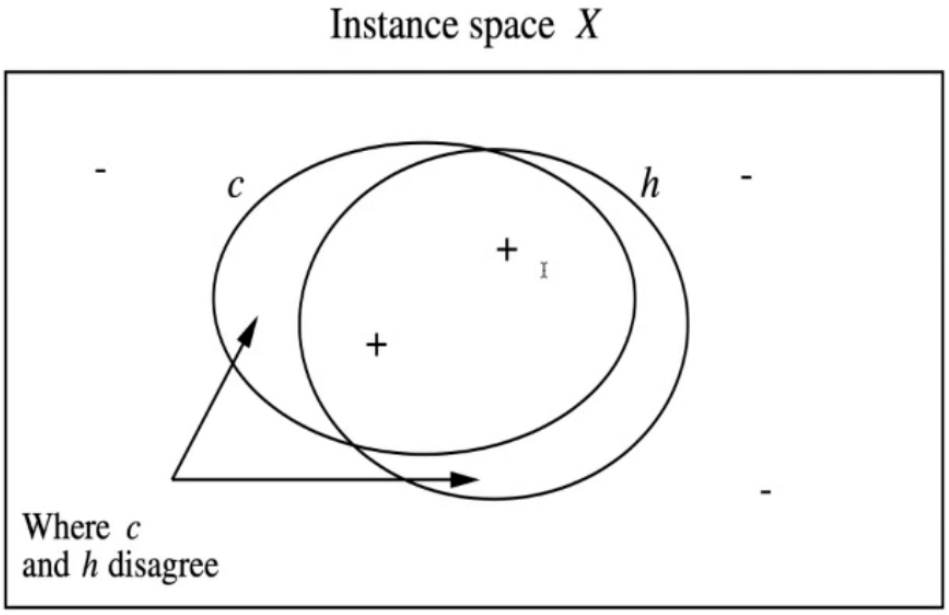
Using this we need to determine a hypothesis h in H, such that h(x) = c(x) for all x in D, and in a more general sense a hypothesis h in H, such that h(x) = c(x) for all x in X (instance space all possible instances).

Our learner observes a set of training examples of the form <x, c(x)> for some target concept c in our set of possible target concepts.

Instances x are drawn from distribution D, and our teacher provides a target value c(x) for each instance.

Using this, our learner must output a hypothesis h that estimates c, where h is evaluated by its performance on subsequent instances drawn according to our distribution D.

## True error of our hypothesis




Here we can see two kinds of errors in our hypothesis space

1. Where c classifies something as positive and our hypothesis h misses it
2. Where h classifies something as positive but our target function c doesn't

**Definition:** *The* **true error** *(denoted* $error_{\mathcal{D}}(h)$*) of hypothesis* $h$ *with respect to target concept* $c$ *and distribution* $\mathcal{D}$ *is the probability that* $h$ *will misclassify an instance drawn at random according to* $\mathcal{D}$*.*

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$$

The true error is just the probability of misclassification (the disagreement in our venn diagram.

### Error

There are two types of error for our hypothesis

1. Training error of our hypothesis h with respect to our target concept c
    a. How often is h(x) =/= c(x) over training set
2. True error of hypothesis h with respect to our target concept c
    a. How often is h(x) =/= c(x) over feature random instances (test set)

Since we cannot really know the true error until testing we need to ask, can we bound our true error of our hypothesis given the training error of our hypothesis? To do this we work under the assumption that our data is noise free. First we consider when the training error of our hypothesis h is zero (our hypothesis is within the version space, where version space is just the set of hypotheses on our data that make no mistakes).

## Concept Learning as search

We want to ask what can be learned, and our answer is only what is in our hypothesis space. For the example above, we need to compute the hypothesis space. To do that first we compute the instance space

Instance space

$$\begin{aligned} Sky \times AirTemp \times \ldots \times Forecast &= 3 \times 2 \times 2 \times 2 \times 2 \times 2 \\ &= 96 \end{aligned}$$

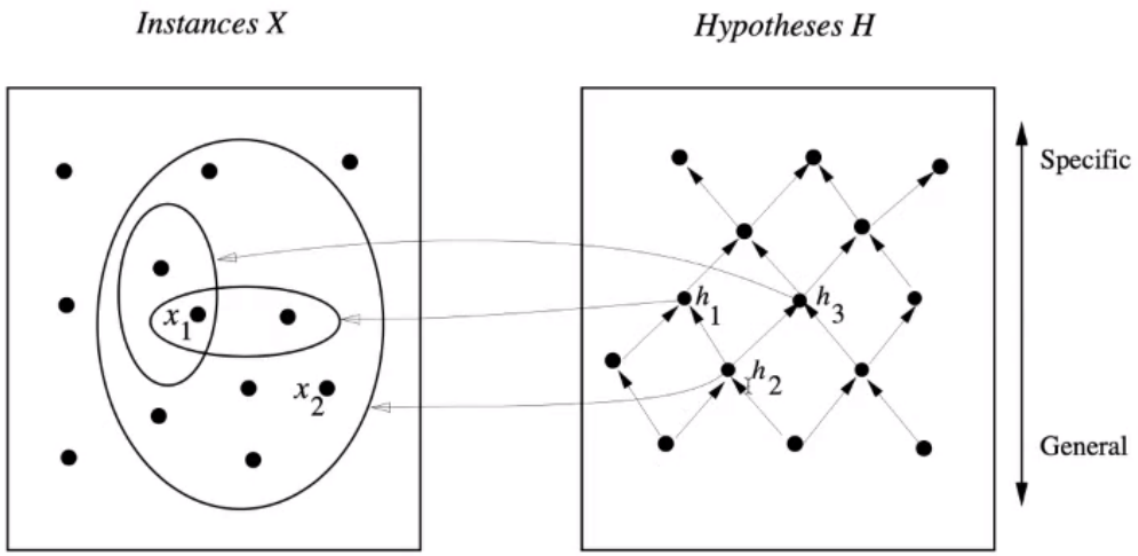
Where the number represents how many discrete values there are for each feature (e.g. hot/cold = 2). Using this we also know our hypothesis space is the following

$$\begin{aligned} Sky \times AirTemp \times \ldots \times Forecast &= 5 \times 4 \times 4 \times 4 \times 4 \times 4 \\ &= 5120 \\ \text{(semantically distinct}^1 \text{ only)} &= 1 + (4 \times 3 \times 3 \times 3 \times 3 \times 3) \\ &= 973 \end{aligned}$$

This is because for our first feature we have 3 values, a case where we don't care about other features, and one where it is missing (3 + 1 + 1) = 5, similarly applies to other features. This is also semantically equivalent to the third line since the constant null error we assume applies to all instances.

Now our learning problem is searching our hypothesis space. We start with our most specific hypothesis, and as examples come in, we generalise by removing specificality. The diagram below shows how hypothesis 1 and 3 are good starting points but won't cover all instances,

however hypothesis 2 covers both instances correctly. We started from more specific hypothesis and generalised




$x_1$= <Sunny, Warm, High, Strong, Cool, Same>
$x_2$= <Sunny, Warm, High, Light, Warm, Same>

$h_1$= <Sunny, ?, ?, Strong, ?, ?>
$h_2$= <Sunny, ?, ?, ?, ?, ?>
$h_3$= <Sunny, ?, ?, ?, Cool, ?>

## Generality in our hypothesis

We can formally define generality on our hypothesis in the following.

If we have h1 and h2 being boolean-valued functions defined over our instances X, then h1 is more general than or equal to h2 IF AND ONLY IF:

For all instances x in our instance space X, h1(x) = 1 implies h2(x) = 1

So if an instance satisfies h2 it also satisfies h1.

## Version Space

A hypothesis h is consistent with a set of training examples D (belongs to the version space) of target concept c if and only if h(x) = c(x) for each training example of our training example in D.

$$Consistent(h, D) \equiv (\forall \langle x, c(x) \rangle \in D)\ h(x) = c(x)$$

Our version space with respect to the hypothesis space H and training examples D, is the subset of hypothesis in H, consistent with all training examples in D.

$$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

Exhausting the version space

The version space is said to be epsilon exhausted with respect to c and D, if every hypothesis in our version space has error less than epsilon with respect to c and D

$$(\forall h \in VS_{H,D})\ error_{\mathcal{D}}(h) < \epsilon$$

So conversely, we can say a version space is not epsilon exhausted if it contains at least one hypothesis h with a higher error than epsilon.

*If the hypothesis space $H$ is finite, and $D$ is a sequence of $m \geq 1$ independent random examples of some target concept $c$, then for any $0 \leq \epsilon \leq 1$, the probability that the version space with respect to $H$ and $D$ is not $\epsilon$-exhausted (with respect to $c$) is less than*

$$|H|e^{-\epsilon m}$$

This lets us know quantitatively, the probability that our hypothesis space is not epsilon exhausted. This theorem allows us to put a bound on the probability that any consistent learner will output a hypothesis h with error(h) >= epsilon.

If we want our bound that our version space is not epsilon exhausted to be below a threshold delta.

$$|H|e^{-\epsilon m} \leq \delta$$

Rearranging we get the following property

$$m \geq \frac{1}{\epsilon}(\ln|H| + \ln(1/\delta))$$

This actually tells us how many examples will epsilon exhaust the version space. We know the minimum number of examples to assure with probability (1 - delta) that every hypothesis in the version space will satisfy our epsilon exhaust property.

Example

Suppose our hypothesis space contains conjunctions of constraints on up to n boolean attributes (n boolean literals, any hypothesis can contain a literal, or its negation or neither).

Then we know our hypothesis space size is 3^n. Using that we can say the following about sample size.

$$m \geq \frac{1}{\epsilon}(\ln 3^n + \ln(1/\delta))$$

$$m \geq \frac{1}{\epsilon}(n \ln 3 + \ln(1/\delta))$$

Using our sports example from before we knew the hypothesis space was 973 large, so for the enjoysport model, we can say the following about the sample size

$$m \geq \frac{1}{\epsilon}(\ln 973 + \ln(1/\delta))$$

If we want to assure that for our playsport example, there is a 95% probability that our version space contains only hypothesis with error that is less than 10%, it is sufficient to have m examples for our training set, where

$$m \geq \frac{1}{0.1}(\ln 973 + \ln(1/0.05))$$

(the 0.05 is the delta (95%) delta is 1 - probability).
This simplifies to, we need

M >= 98.8 samples.

Bringing it all together

Consider a class $C$ of possible target concepts defined over a set of instances $X$ of length $n$, and a learner $L$ using hypothesis space $H$.

> Definition: $C$ *is* **PAC-learnable** *by* $L$ *using* $H$ *if for all* $c \in C$, *distributions* $\mathcal{D}$ *over* $X$, $\epsilon$ *such that* $0 < \epsilon < 1/2$, *and* $\delta$ *such that* $0 < \delta < 1/2$, *learner* $L$ *will with probability at least* $(1 - \delta)$ *output a hypothesis* $h \in H$ *such that* $error_{\mathcal{D}}(h) \leq \epsilon$, *in time that is polynomial in* $1/\epsilon$, $1/\delta$, $n$ *and* $size(c)$.

Note this is all under the assumption of discrete noise-free data.

## Real-Valued hypothesis space

Till now we have been working on discrete valued attributes. in a real valued hypothesis space, a ray is a one-dimensional threshold function defined for a real valued theta by

$$r_\theta(x) = 1 \iff x \geq \theta$$

To find these rays, we want to find the smallest ray containing all positive examples in the training set.

Given the following training examples

$$D = \{\langle x_1, c(x_1)\rangle, \langle (x_2, c(x_2)\rangle, \ldots \langle x_m, c(x_m)\rangle\}$$

The hypothesis output of the learner should be r(lambda) where lambda is the following

$$\lambda = \arg\min_{1 \leq i \leq m} \{x_i \mid c(x_i) = 1\}$$

And lambda is infinite if the training set has no positive examples of the concept.

We are just looking for the smallest line which contains all positive examples of our real valued attribute.

The simple memoryless online algorithm to learn rays, based on finding the minimum of a set is as follows

1. Set lambda to infinity

2. Iterate over the training set
    a. If the classifier gives us a positive example for the instance, and the instance has a smaller value than lambda, we set lambda to the instance
3. Finally L(D) our learner is just = r(lambda)

### Example

Given the following training set

$$\{\langle 3.6578, 1\rangle, \langle 2.5490, 0\rangle, \langle 3.4156, 1\rangle, \langle 3.5358, 1\rangle, \langle 3.3413, 1\rangle, \langle 4.4987, 1\rangle\}$$

The corresponding sequence of hypothesis is as follows

$$r_{\infty},\ r_{3.6578},\ r_{3.6578},\ r_{3.4156},\ r_{3.4156},\ r_{3.3413},\ r_{3.3413}$$

If the training set is for some target concept r(theta), then the hypothesis returned by the learning algorithm should be a ray r(lambda) where lambda >= theta.
As the size of the training set increases, we should expect a smaller error using our hypothesis r(lambda) over r(theta). This basically says our error becomes reduced as the training set increases. The error can be seen as (lambda - theta).

## Noise in our learner

So far we have assumed that we have consistent learners (noise free),

$$c \in H$$

This is us saying that our true function c which perfectly classifies is representable, but this isn't always the case and a strong assumption to make, we want to relax this assumption. We want to now find the hypothesis h that makes the fewest error on training data. Similar to before the sample complexity is now.

$$m \geq \frac{1}{2\epsilon^2}(\ln |H| + \ln(1/\delta))$$

(our sample size is increased since the denominator is a value < 1 that is squared). This is all derived from hoeffding bounds which are used in practical learning methods.

## Unbiased Learners

An unbiased concept class C, contains all target concepts definable on instance space X where

$$|C| = 2^{|X|}$$

This means any subset of the data can define the concept. Here we are relating the size of our concept class to the size of the instance space.

For example suppose our instance space is defined by using n amount of features all boolean valued, then we know the size of our instance space is 2^n meaning

$$|C| = 2^{2^n}$$

(all the ways we can label our instance space).

An unbiased learner has a hypothesis space able to represent all possible target concepts,
H = C.
We can tell from this that an unbiased learner will have the following property on the number of samples

$$m \geq \frac{1}{\epsilon}(2^n \ln 2 + \ln(1/\delta))$$

What we can see is that this is bad since the samples required grows exponentially with the number of features. This tells us it is good to have a good inductive bias for learning as this approach is computationally expensive.

Unbiased learning is too hard, we need a huge amount of instances to make an unbiased learner viable assuming we have infinite computation power.

## VC Dimension

VC dimensions help us answer the question, how complex is a hypothesis?

## Shattering a set of instances

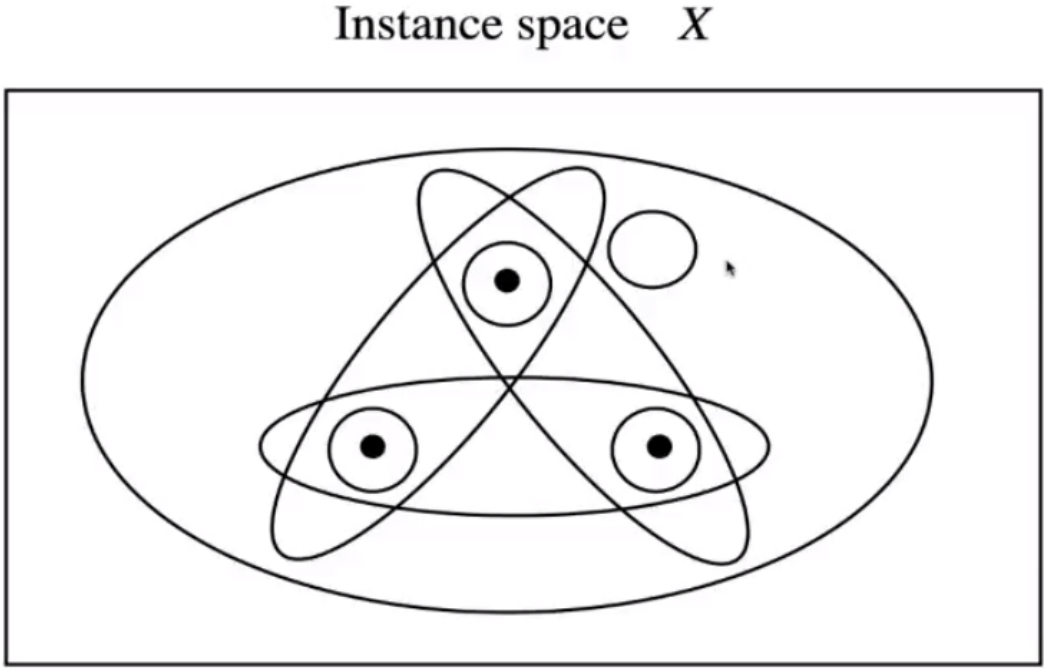
Shattering is a classification idea. The definition is given as the following:

A dichotomy of a set S is a partition of S into two disjoint subsets. A → B and C where B U C = A

A set of instances S is shattered by hypothesis space H if and only if for every dichotomy of S, there exists some hypothesis in H consistent with this dichotomy.

For each hypothesis there is a dichotomy, and for every dichotomy there is a hypothesis.

Example below shows a 3 instance shatter, where each hypothesis is represented by a dot, and each dichotomy is represented with a circular enclosing



## Example

Suppose we have the following instances

$$n = \text{ManyTeeth} \wedge \neg\text{Gills} \wedge \neg\text{Short} \wedge \neg\text{Beak}$$
$$g = \neg\text{ManyTeeth} \wedge \text{Gills} \wedge \neg\text{Short} \wedge \neg\text{Beak}$$
$$s = \neg\text{ManyTeeth} \wedge \neg\text{Gills} \wedge \text{Short} \wedge \neg\text{Beak}$$
$$b = \neg\text{ManyTeeth} \wedge \neg\text{Gills} \wedge \neg\text{Short} \wedge \text{Beak}$$

Here we can see in our data set we have 16 possible subsets for the set {n, g, s, b} (power set = 2^4)

We can represent each subset by its own conjunctive concept by taking the negated literal in the conjunction of all clauses for example, we can say

- Subset {n} = !gills and !short and !beak
- Subset {n, g, s} = !gills and !short
- Subset {g, s} = !manyteeth and !beak

So on and so on, we can do this for all 16 subsets

This allows us to say that this set of four instances is shattered by the hypothesis language of conjunctive concepts. (so the language of conjunctions shatters this instance space since each subset can be represented by a hypothesis of conjunctions).

## Generalisation

In a general sense for shattering, if we have a dataset described by d amount of boolean features, and a hypothesis space of conjunctions up to d boolean literals, then the largest subset of instances that can be shattered is at minimum d.

### Example

Suppose d = 3, and our instance space is represented by having the i-th boolean literal set to true for instance i, and the rest set to 0

$$\text{instance}_1: \ 100$$
$$\text{instance}_2: \ 010$$
$$\text{instance}_3: \ 001$$

Now any dichotomy can be constructed by a conjunctive hypothesis, similar to before example, that excludes any instance simply by adding the appropriate literal, for example the hypothesis

only including instance 2 is the one that says !attr1 and !attr3. In fact we can say that the largest subset of instances that can be shattered in this setting has size exactly d.

## Definition of Vapnik-Chervonenki's dimension

The definition of our VC dimension, VC(H), of the hypothesis space H which is defined over our instance space X, is the size of the largest finite subset of X shattered by H. if arbitrary large finite sets of X can be shattered by H then we say the VC(H) = infinity.

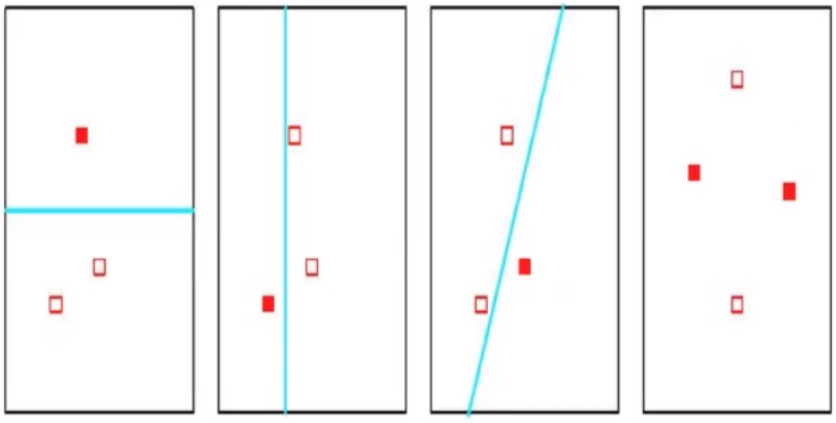
In the previous example, VC(H) = 3. In Fact if we have an instance space X which is constructed from d amount of boolean features, and a hypothesis space H of conjunctions of up to d boolean literals, then the VC dimension will always be VC(H) = d.

VC dimensions can be defined for an infinite hypothesis space H, since it only depends on the size of the finite subsets of X not the hypothesis space.

This is a way to define the complexity of a hypothesis space, in terms of our instance space X.

### VC dimension of linear decision surfaces

Consider that we have linear classifiers in two dimensions, how do we compute the VC dimension of this class of hypothesis. From the diagram below we can clearly see if we have a subset of 2 instances, we can find a linear classifier for all possible dichotomies



Having a subset of 1, 2 and 3 instances for a linear classifier, we can linearly separate all examples into dichotomies, however when we added a 4th instance we run into the XOR problem (see right most one) it cannot be linearly separated, so we ran into a case where 4 instances for 2 features, was unable to be linearly separated. This means that our VC dimension is 3 (d + 1) since that is the largest possible finite set in which we can apply our hypothesis to all cases.

## Sample complexity from VC

We can now generalise pac-learning results obtained earlier to answer the following question, how many randomly drawn examples would be enough to epsilon exhaust our version space with probability of at least (1-delta)?
The following sample complexity is:

$$m \geq \frac{1}{\epsilon}(4\log_2(2/\delta) + 8VC(H)\log_2(13/\epsilon))$$

We are able to use the VC dimension of our hypothesis space, to substitute in the case where we have an infinite hypothesis space (powerful stuff).

So we see that the concept of the VC dimension of a hypothesis class gives us a general framework for characterising the complexity or *capacity* of hypotheses in terms of their ability to express all possible target concepts in a particular learning setting.

For example, the argument developed for the VC dimension of linear classifiers can be extended to multi-layer perceptrons to obtain sample complexity bounds in the PAC-learning framework.

## Mistake convergence

Up until now we saw how many examples were needed to learn, but now we want to know how many mistakes before we get convergence? We want to see if it's possible to bound the number of mistakes the learner makes before converging. We can start by similarly analysing concept learning.

Similar to PAC

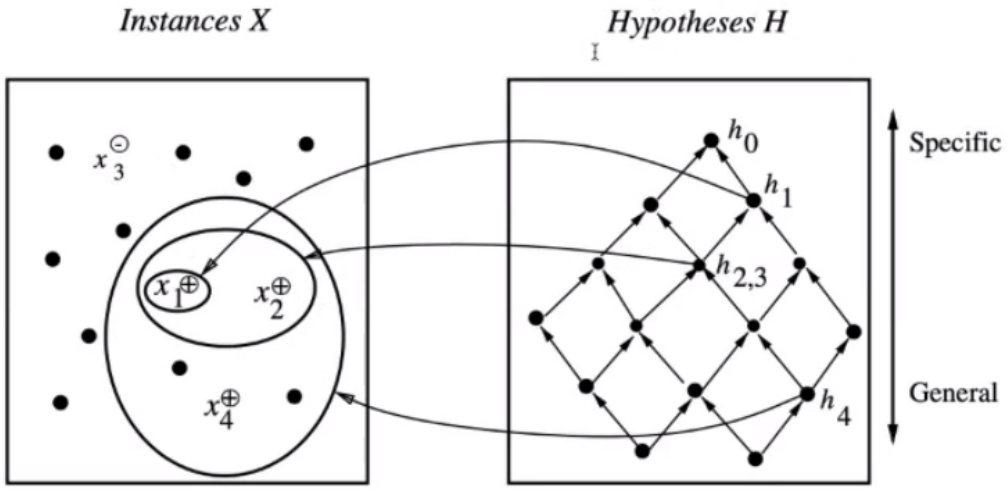
- Instances drawn at random from $X$ according to distribution $\mathcal{D}$
- Learner must classify each instance before receiving correct classification from teacher

## Find-S Algorithm

An online specific-to-general concept learning algorithm is the following

- Initialize h to the most specific hypothesis in H
- For each positive training instance x
    - For each attribute constraint ai in h
        - If the constraint is satisfied by x -> do nothing
        - Else replace our attribute constraint by the next more general constraint satisfied by x.

An example of the algorithm is as below




$h_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$

$x_1 = \langle Sunny\ Warm\ Normal\ Strong\ Warm\ Same \rangle, +$ $h_1 = \langle Sunny\ Warm\ Normal\ Strong\ Warm\ Same \rangle$

$x_2 = \langle Sunny\ Warm\ High\ Strong\ Warm\ Same \rangle, +$ $h_2 = \langle Sunny\ Warm\ ?\ Strong\ Warm\ Same \rangle$

$x_3 = \langle Rainy\ Cold\ High\ Strong\ Warm\ Change \rangle, -$ $h_3 = \langle Sunny\ Warm\ ?\ Strong\ Warm\ Same \rangle$

$x_4 = \langle Sunny\ Warm\ High\ Strong\ Cool\ Change \rangle, +$ $h_4 = \langle Sunny\ Warm\ ?\ Strong\ ?\ ? \rangle$

We start off with the most specific hypothesis, where everything is null, we receive our first data x1, and we simply can set the constraints to be all the attributes of x1. Now when we receive x2, we can see that the 3rd attribute does not line up and classify both correctly, so we have to take the more general hypothesis of ignoring 3rd attribute. X3 comes in as a negative example so we don't care next, and finally example 4 comes in, and we see that the current hypothesis is too specific and will misclassify, so we need to change the hypothesis to ignore the last 2 attributes in order to correctly classify everything.

Assuming a hypothesis h in our hypothesis space describes our target function c, and that training data is error-free (no noise). By definition h is consistent with all positive training examples so we never have to cover a negative example.

For each hypothesis hi generated Find-S, h is more general than or equal to hi. So this means hi will never cover a negative example, simply by the property of more general than or equal.

### Issues in FIND-S

1. Can't tell whether it has learned the concept
    a. Learned hypothesis is a part of the version space, but may not be the only consistent hypothesis.
2. Can't tell when the training data is inconsistent, it cannot handle noise/misclassification
3. Strong inductive bias that it will always pick the most specific hypothesis
    a. Sometimes we require the most general hypothesis

## Example Find-S mistake bound

Consider FIND-S when our hypothesis space is just a conjunction of boolean literals

FIND-S:

- *Initialize* $h$ *to the most specific hypothesis* $l_1 \wedge \neg l_1 \wedge l_2 \wedge \neg l_2 \ldots l_n \wedge \neg l_n$
- *For each positive training instance* $x$
    - *Remove from* $h$ *any literal that is not satisfied by* $x$
- *Output hypothesis* $h$.

Now we can ask how many mistakes before we converge to the correct hypothesis h. Find-s will converge to error-free hypothesis in the limit if

1. Concept class is within the Hypothesis space
2. Data is noise free

FIND-S will classify all instances as negative at first. It will generalise for positive examples by dropping unsatisfied literals. Since our target function is within our hypothesis space, we will never classify a negative instance as positive. The mistake bound then comes from the number of positives classified as negatives.

First

1. We have 2n terms in initial hypothesis (example if our attributes were just A B)
    a. Initial hypothesis is (A, !A, B, !B)
2. The first mistake will remove half these terms leaving us just n, since each instance can only have 2 attributes (ex. A = true, B = false)
    a. New hypothesis (A, !B)
3. Each subsequent mistake will remove at least 1 term to generalise to a hypothesis to cover all positive examples.

4. In the worst case we would remove all n-remaining terms
    a. Everything is positive (trivial)
5. Worst case number of mistakes would then be n + 1
6. Worst case sequence of learning steps, removing one literal per step, each step is a mistake

Final answer, mistake bound (n + 1)

### Halving Algorithm

FIND-S returns the most specific consistent hypothesis. An extension of the FIND-S concept learning algorithm is the candidate-elimination algorithm which returns all consistent hypothesis, giving us the version space.

Now we can consider a different algorithm “Halving Algorithm”

- learns concepts using candidate-elimination algorithm
- Classifies new instances by majority vote of the version space hypothesis

Simply put, it will check the version space hypothesis’, and it will classify instances by a majority vote among the version space.

The halving algorithm learns once the version space contains only one consistent hypothesis which corresponds to the target concept

1. Take majority vote amongst verison space
2. All hypothesis in version space which are in the minority is removed
3. Mistake occurs when majority vote classification is incorrect.

In the worst case, on every step, mistake because majority vote is always incorrect, each mistake causes the number of hypotheses to be reduced by at least half. The worst case mistake bound is then seen as

$$\lfloor \log_2 |H| \rfloor$$

In the best case, on every step no mistake because the majority vote is correct, still remove all incorrect hypotheses up to half, in best case 0 mistakes in converging to correct hypotheses.

## WINNOW algorithms

Mistake bounds are based on the work of Nick Littlestone who developed the WINNOW class of algorithms.

- Online mistake-driven algorithm
    - Similar to the perceptron
    - Learns linear threshold function

- Designed to learn in the presence of many irrelevant features
    - The number of mistakes grow only logarithmically with the number of irrelevant features
- We will work with WINNOW2
    - WINNOW1 we eliminate attributes not demote them.

## The WINNOW2 Algorithm

While some instances are misclassified
  For each instance $x$
    classify $x$ using current weights $w$
    If predicted class is *incorrect*
      If $x$ has class 1
        For each $x_i = 1$, $w_i \leftarrow \alpha w_i$ # Promotion
        (if $x_i = 0$, leave $w_i$ unchanged)
      Otherwise
        For each $x_i = 1$, $w_i \leftarrow \frac{w_i}{\alpha}$ # Demotion
        (if $x_i = 0$, leave $w_i$ unchanged)

Here $x$ and $w$ are vectors of features and weights, respectively.

Here we see, if we misclassify and we get a negative when we should have had a positive, we want to see which features for that instance impact the prediction, and promote their weighting. In the other case if we have a negative example classified as positive, we want to do the same except decrease the weighting.

- The user supplies threshold theta such that for any instance a
    - class is 1 if $\sum w_i a_i > \theta$
- Similar to perceptron training rule
    - Alpha > 1
    - Winnow2 uses multiplicative weight updates
- Will do much better than perceptron with many irrelevant attributes as the number of attributes does not really affect the efficiency.
- To learn the irrelevant attributes
    - Assume that the target concept can be expressed using a finite subset of the attributes

- If there are n attributes in total, n - r attributes are considered irrelevant (all attributes not used in the final model)
- Mistake-bounds for winnow type algorithms are logarithmic in the number of irrelevant attributes
- Worst cause mistake bound is roughly O(rlog(n)).

Might be useful idk

$$VC(C) \leq Opt(C) \leq M_{Halving}(C) \leq log_2(|C|).$$

## Weighted Majority

This is a generalisation of the halving algorithm idea.

- Predicts by weighted vote of set of prediction algorithms
- Learns by altering weights for prediction algorithms
- Any prediction algorithm simply predicts value of target concept given an instance, they can be
    - Elements of the hypothesis space H or even different learning algorithms
- If there is inconsistency between prediction algorithm and training example then we reduce the weight of prediction algorithm
- Bound number of mistakes of ensemble by the number of mistakes made by the best prediction algorithm
    - Pick the cream of the crop.

This is a powerful ensemble method.

The algorithm

$a_i$ is the $i$-th prediction algorithm
$w_i$ is the weight associated with $a_i$

For all $i$, initialize $w_i \leftarrow 1$
For each training example $\langle x, c(x) \rangle$
  Initialize $q_0$ and $q_1$ to 0
  For each prediction algorithm $a_i$
    If $a_i(x) = 0$ then $q_0 \leftarrow q_0 + w_i$
    If $a_i(x) = 1$ then $q_1 \leftarrow q_1 + w_i$
  If $q_1 > q_0$ then predict $c(x) = 1$
  If $q_0 > q_1$ then predict $c(x) = 0$
  If $q_0 = q_1$ then predict 0 or 1 at random for $c(x)$
  For each prediction algorithm $a_i$
    If $a_i(x) \neq c(x)$ then $w_i \leftarrow \beta w_i$

Here q0 and q1 are our counters for people who classify either 0 or 1 respectively. This is similar to the halving algorithm, we downweight the prediction algorithms which predicted in the minority.

Key result: number of mistakes made by WEIGHTED MAJORITY algorithm will never be greater than a constant factor times the number of mistakes made by the best member of the pool, plus a term that grows only logarithmically in the number of prediction algorithms in the pool.

## Limitations to machine learning

We want to answer the following questions

1. Are there reasons to prefer one learning algorithm to another
2. Can we expect any method to be the best for the task
3. Can we even find an algorithm that is overall superior to random guessing?

The answer to all of them is no (sad face)

Unless! We have prior knowledge on or can make assumptions about the distribution of target functions. This is a consequence  of a number of results known as the “No Free Lunch Theorem.”

Since the results specifically address the issue of generalising from a training set to minimise off-training ser errors, they are referred to as “conservation laws of generalisation”.

## No Free Lunch Theorem

Two main results of this theorem are

- Uniformly averaged over all target functions, the expected off-training-set error for all learning algorithms is the same
- Assuming the training set D can be learned correctly by all algorithms, averaged over all target functions, no learning algorithm gives an off-training set error superior to any other

$$\Sigma_F[\mathbb{E}_1(E|F,\mathcal{D}) - \mathbb{E}_2(E|F,\mathcal{D})] = 0$$

where $F$ is the set of possible target functions, $E$ is the off-training set error, and $\mathbb{E}_1$, $\mathbb{E}_2$ are expectations for two learning algorithms.

## Example

We are given this setup

- Consider a data set with three Boolean features, labelled by a target function $F$
- Suppose we have two different (deterministic) algorithms that generate two different hypotheses, both of which fit the training data $\mathcal{D}$ exactly
- The first algorithm assumes all instances $x$ are in the target function $F$, unless labelled otherwise in training set $\mathcal{D}$, and the second algorithm assumes the opposite
- For *this particular target function* $F$ the first algorithm is clearly superior in terms of off-training-set error
- But this cannot be determined from the performance on training data $\mathcal{D}$ alone !

And here is the example

| | $x$ | $F$ | $h_1$ | $h_2$ |
|---|---|---|---|---|
| $\mathcal{D}$ | 000 | 1 | 1 | 1 |
| | 001 | -1 | -1 | -1 |
| | 010 | 1 | 1 | 1 |
| | 011 | -1 | 1 | -1 |
| | 100 | 1 | 1 | -1 |
| | 101 | -1 | 1 | -1 |
| | 110 | 1 | 1 | -1 |
| | 111 | 1 | 1 | -1 |

So h1, gives everything positive unless the training example has an exact example to disagree, and h2 does the opposite.

We can see that despite same performance on the training set

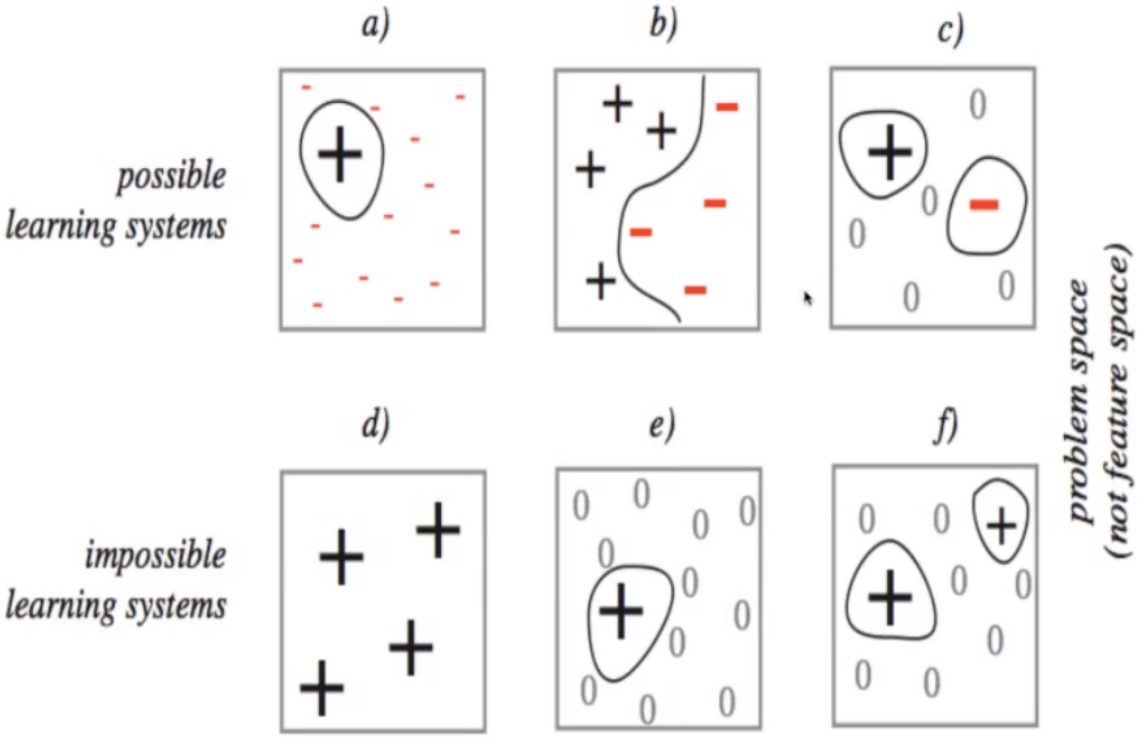
1. Error on h1 = 40%
2. Error on h2 = 60%

The point of this is that, unless we know something about F, we can't really have an inductive bias or say method A is better than method B. there are 2^ target functions consistent with the data, for each of these target functions there is exactly one other function whose output is the inverse with respect to each of the off-training instances, so performance of algorithm 1 and 2 will be inverted and the error difference will cancel. This ensures the average error difference is 0.

For every possible learning algorithm for binary classification the sum of performance over all possible target functions is exactly zero !

- on some problems we get positive performance
- so there *must* be other problems for which we get an *equal and opposite amount* of negative performance

Sometimes we get lucky picking our model, other times we don't, we can't tell how it would perform on a different test set in comparison to another model WITHOUT KNOWING THE INDUCTIVE BIAS BEHIND THE DATA OR DISTRIBUTION.




We can't just have all positives, it must balance out. It is impossible to have good performance all around with no bad performance, in the possible systems, the good is cancelled out by the negative, really good is cancelled by a lot of smaller bads, average good is cancelled by average bad etc. no one algorithm will fit the impossible systems on the 2nd row.

## Summary

Algorithm independent analyses using techniques from computational complexity, pattern recognition and statistics have led to the discovery of many new techniques. Some results are over-conservative from a particle view points, they look at the worst-case rather than the average-case analysis.

These algorithms came out of the following theories

1. Boosting algorithm
    a. Originated from the idea and answering questions on PAC learning
2. SVM
    a. Originated from the theory of VC dimensions
3. WINNOW, WINNOW2, Weighted Majority

    a. Originated from mistake bound theory.
4. Ensemble learning methods
    a. Came from the idea of bias-variance decomposition.

# The Takeaway, most important part imo for working as a ML engineer

Even the popular, theoretically well founded algorithms for example, Naive Bayes, will perform poorly on some domains, in cases where the algorithm is not a good match to the class of the target function.what we aim to find is the assumptions behind the learning domains (the inductive bias) which helps us pick the algorithm for the job.

Since there is no free lunch, having an experience with a broad range of techniques is the best insurance for solving arbitrary new classification problems. (see kaggle problem top place submission had done over 31 learning algorithms to even understand the data, and more

processing) you need a good understanding of each algorithm's weaknesses, strengths and what you can get out of its performance to become a great ML engineer.