IAML Condensed Summary Notes For Quick In-Exam Strategic Fact Deployment

Maksymilian Mozolewski

December 13, 2020

Contents

IAML

Introduction

Machine Learning

A machine learning model **takes in** data, **outputs** predictions. It's a function of data really together with a set of training data.
Learning = Representation + Evaluation + Optimisation

Thinking about data

Classification

Sort data points into discrete buckets based on training data

Regression

Output a continuous/real value for each data point based on training data.

Clustering

Detect which data points are related to which other data points, find outliers.

Data representation

What format do we feed the data in ? Most likely as a **bag of features**. I.e. collection of attribute-value pairs, every data point must have an attribute-value pair for each property (in most cases)
Data representation has more impact on the performance of your ML algorithm than anything.

Types of attributes

- **Categorical**
  - e.g. red/blue/brown
  - a set of possible **mutually exclusive** values
  - meaningful operators: equality comparison
  - usually represented as numbers
  - problems: synnonymy is a major challenge e.g. some values might mean the same thing to a human but not to the machine (country == folk?)
- **Ordinal**
  - e.g. poor < satisfactory < good < excellent
  - a set of possible **mutually exclusive** values, but with a **natural ordering**
  - meaningful operators: equality comparison, sorting
  - problems: sometimes hard to differentiate from categorical (single < divorced)?
- **Numeric**
  - e.g. 3.1/5
  - meaningful operators: arithmetic, distance metrics, equality, sorting
  - problems:
    - sensitive to extreme outliers (handle these **before normalization**)
    - skewed distributions (assymetric) - outliers might actually be real data (e.g. personal wealth data)
    - Non-monotonic effect of attributes - e.g. predicting someone is going to win a marathon, here the relationship is not monotonic i.e. norect correlatio din, might be a curve with a "sweet spot"
  - solutions:
    - *Deal with outliers, maybe trim them for training phase only?*
    - *use a log/atan scale to make data more linear*
    - *discretize data into buckets*

Normalisation

Normalization is the process of converting all the data such that each different attribute is roughly in the same range, and comparable.
Normalization is mostly necessary for linear methods.

Picking attributes

We want:

- all our attributes to have similar values if the data points that posses them are similar themselves!
- small change in input $\rightarrow$ small change in values

Images

For images, can we use pixel data as attributes directly ? it depends, if the pixel 20,20 always corresponds to the middle of a letter then yes, this is a meaningful attribute which might help us discern whether digits given have strokes going through the middle of the image! What if the pixel is always something random ? This could happen whenever we are looking for an object which can be anywhere in the image, in which case the attribute would be gibberish! In the case of image classification, we want attributes which are:

- invariant to size
- invariant to rotation
- invariant to illumination

How can we classify whether an image contains a desired object ? In general we do the following:

- **Segment** the image, into regions of pixels which we believe are related (by colour, texture, position etc..)
- Pick a number of attributes which you will use to describe each of the regions
- Then use those features in your machine learning algorithm!

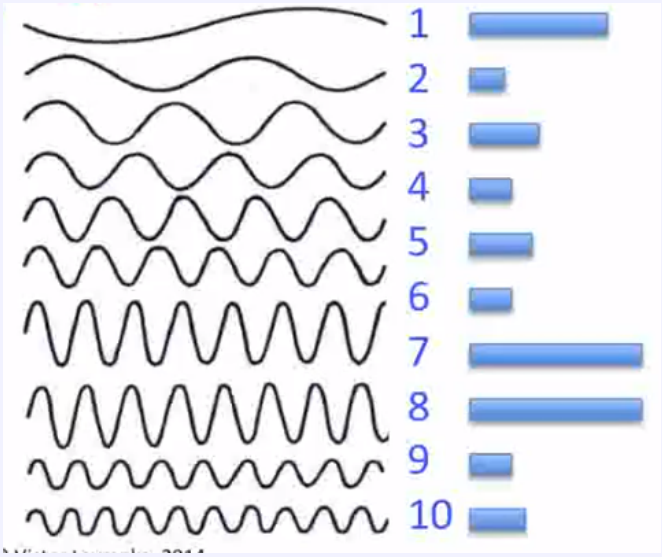
Keep in mind though, the segmentation **will** make errors, we can hope that these will be consistent across all images (all images will have their legs chopped off). Sometimes we can segment using a grid

Text

For textual data, often we can use the **bag-of-words** approach. I.e. we can form an attribute vector which counts the amount of occurrences of each word, regardless of position. This is invariant to word shuffling for example.

Sound

For sound, the data is sound waves. How can we select attributes here? We can count the number of different frequencies occurring in the piece, (using Fourier's transform) and treat this as a feature vector!



Supervised Learning

Supervised learning algorithms have some sort of "performance" metric they can use, i.e. test labels they can validate their guesses on. When the algorithm can measure accuracy directly it's a supervised algorithm.

Unsupervised Learning

Learning without a specific accuracy measure available. Algorithms in this area usually look for structure/patterns/information in the data which can be helpful in other ways. There is nothing specific the algorithm is looking for. Can be **direct** when the algorithm helps to make sense of the data directly, or **indirect** when it is "plugged" into another machine learning aglorithm as an attribute itself.

Semi-supervised Learning

Using unsupervised methods to improve supervised algorithms. Usually have a few labelled examples but lots more unlabelled.

Multi-class classification

Classification with multiple mutually exclusive labels/classes.
Might be hard to tell when something belongs to none of the available classes.

Binary classification

Classification with 2 mutually exclusive labels/classes in each "run". This way of classification can be applied to multiple-classes classification but with a "One-vs-Rest" meta-strategy (a vs not a, b vs not b). In this way a sample may belong to multiple classes but never to two sides of the one-vs-rest structure simultaneously in each run.
In this classification method we can actually tell when something doesn't belong to any class!

Analysing data

We have to check for a number of things in our data sets:

- Are there any dominating classes ? what would the best "dummy" model do ? always predicting no ?
- What should we use as the appropriate error metric ? How important are the false positives vs the false negatives ?

Generative model

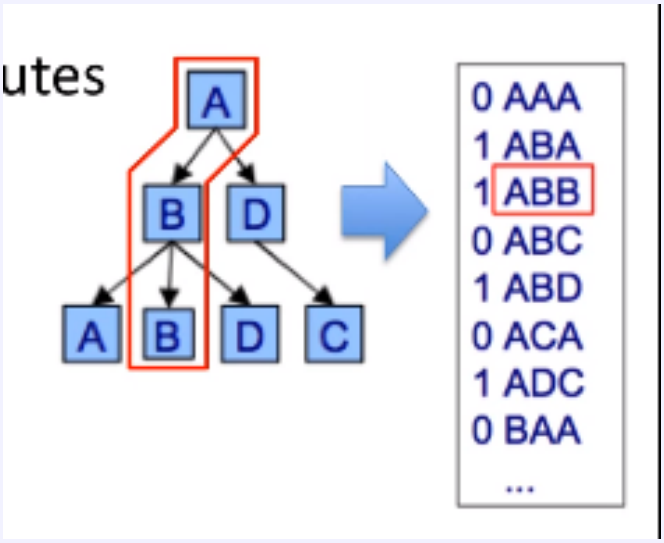
A generative model, develops a probabilistic model of each class, i.e. tries to "model" the underlying probability distribution directly. The decision boundary becomes implicitly defined by the probabilities of each input being in each class.

Discriminative model

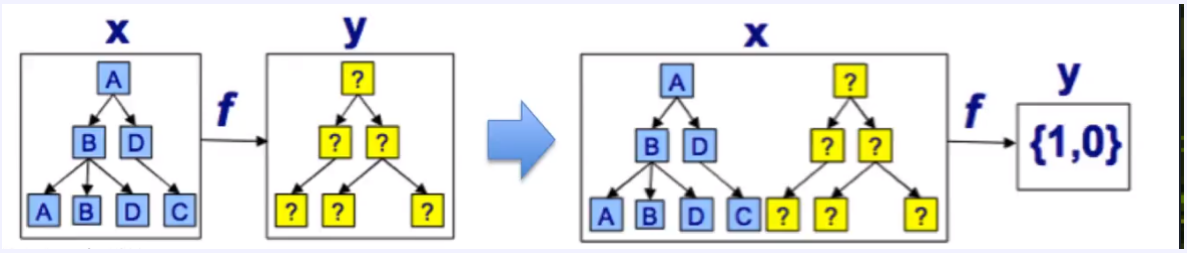
A discriminative model ignores the underlying model and tries to "separate" the data, i.e. it tries to model the boundaries that divide the classes. **Not designed to use unlabeled data** so cannot be used for unsupervised tasks.

Dealing with data structure

What do we do if the data input has some sort of hierarchical structure ? Where the position of occurrence of a node affects its meaning? We can encode as attributes the existence of root-to-leaf paths in the entire tree, and use this bag-of-words approach to perform machine learning



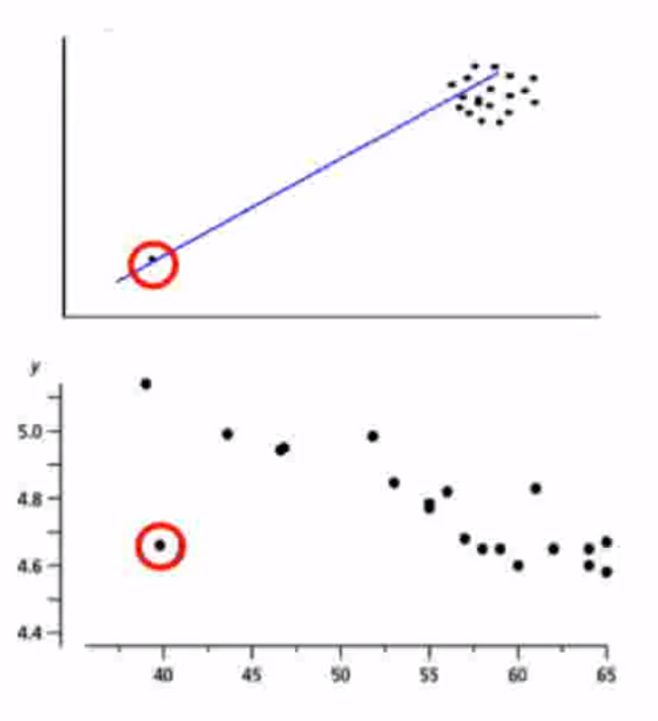
What if we need to predict the output structure from the input structure ? This is very difficult, but we can "trick" our classifier and turn this more into a search problem by embedding the possible outputs with each input and classifying on that instead:



This of course means we have to search for all possible output structures!

Dealing with outliers

Outliers are isolated instances of a class that are unlike any other instance of that class. These affect all learning models to some degree.
There are some ways we can deal with outliers. One method is to remove the outliers just before we perform any sort of normalisation on the data, (ONLY FOR THE PURPOSES OF TRAINING!!) We can also put a confidence interval around our data, and removing values outside of those intervals (with x,y values outside of a normal range). Some data points might still be outliers even though they are within expected x,y ranges! (second figure)



Best way to deal with outliers ? **VISUALISE YOUR DATA**

Naive Bayes

Bayes rule

$$P(y|x) = \frac{P(x|y)P(y)}{\sum_{y'} P(x|y')P(y')} \tag{1}$$

where:

- y : one of the classes
- x : the input data, feature vector
- P(y) : **prior**, the probability of seeing elements of class y in general prior to making any observations
- $P(x|y)$: **class model/likelihood**, given the data is in class y, how likely are the features that i am seeing
- P(x) $= \sum_{y'} P(x|y')P(y')$: **normalizer**, does not affect the probabilities, but without this term we cannot compare data in terms of probability of being in each class (i.e. for confidence values)

Note: this works with probability densities too (which we're using almost exclusively)

Naive bayes

Bayesian classifier, which assumes **conditional independence** between different attributes. It attempts to model the underlying probability distribution so it's a **generative** model.
I.e.:

$$P(\mathbf{x}|y) = \prod_{i=1}^{d} P(x_i|x_1 \dots x_{i-1}, y) = P(x_1|y) \cdot P(x_2|y) \cdot P(x_3|y) \dots \tag{2}$$

It assumes that there are no correlations between the variables themselves, and any correlations are explained by the fact they belong to the same class.

Example

The probability of going to the beach and getting a heat stroke are not independent: $P(B, S) > P(B)P(S)$. The may be independent if we know the weather is hot (i.e. external factor is what actually causes the dependence, which we can factor into the equation): $P(B, S|H) = P(B|H)P(S|H)$ This hidden factor in naive bayes is **the class**

Gaussian Naive bayes

A Naive Bayes classifier which uses the gaussian distribution as a class model for each class.

$$p(x|y) = \frac{1}{\sqrt{2\pi\sigma_{x,y}^2}} \exp^{-\frac{1}{2}\frac{(x-\mu x,y)^2}{\sigma_{x,y}^2}} \tag{3}$$

With:

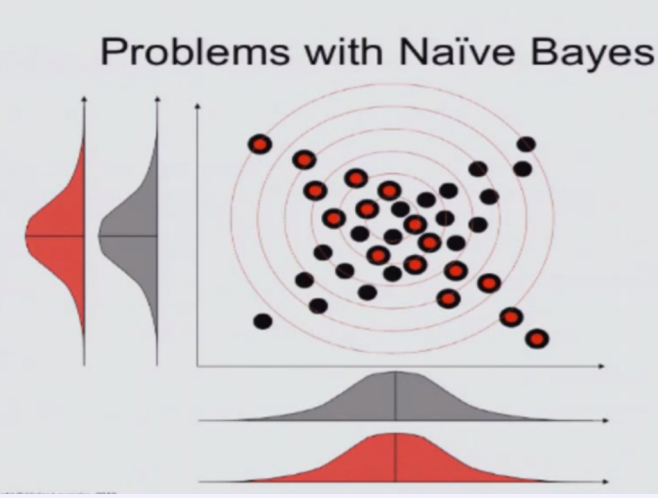
- $\mu_{x,y}$: the mean of the gaussian modelling class y's attribute x
- $\sigma_{x,y}^2$: variance of the gaussian modelling class y's attribute x
- x : the value of the attribute
- y : the class

Note: this is not actually a probability per-se, this is a p.d.f function which can be higher than 1

Problems with Naive Bayes

Covariance

Naive bayes cannot model covariance of the attributes, i.e.:



in the above case the only thing differentiating the classes is their relationship between x and y, i.e. the two attributes. Naive bayes can only model **spherical** distributions

Zero frequency problem

Should any data never appear under any of the classes the probability of it belonging to that class is always going to be zero. So say we are classifying emails as spam or non-spam, and the word "now" never occurred in non-spam emails, but had occurred in some spam emails. Any sentence containing the word "now" would always be classified as spam no matter the actual probability of it being spam!
Solution, add a small epsilon to all "feature counts", or perform laplace smoothing:

$$P(w,c) = \frac{num(w,c) + \epsilon}{num(c) + 2\epsilon} \tag{4}$$

Zipf's law: 50% of words occur once

Independence assumption

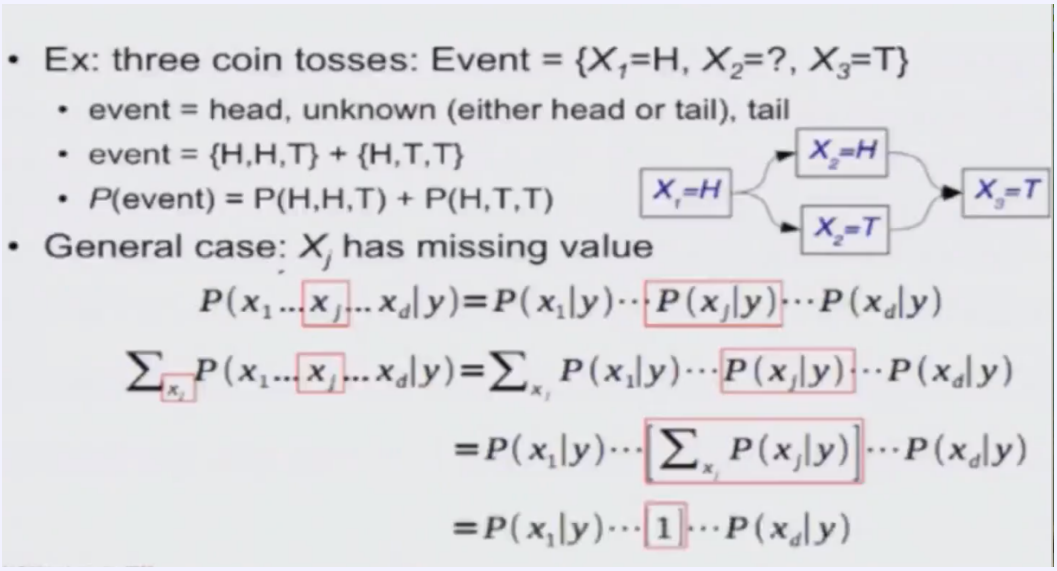
Continuing with the spam example, every word contributes to the total probability independently, easy to fool. Simply stuff lots of non-spammy words into email

Deaing with missing data

Suppose we have missing data for some attribute i of a sample x. How can we compute the likelihood of this sample belonging to any class ?
We simply ignore it:

$$P(x_1 \dots x_i \dots x_d|y) = \prod_{k \neq i}^{d} P(x_i|y) \tag{5}$$

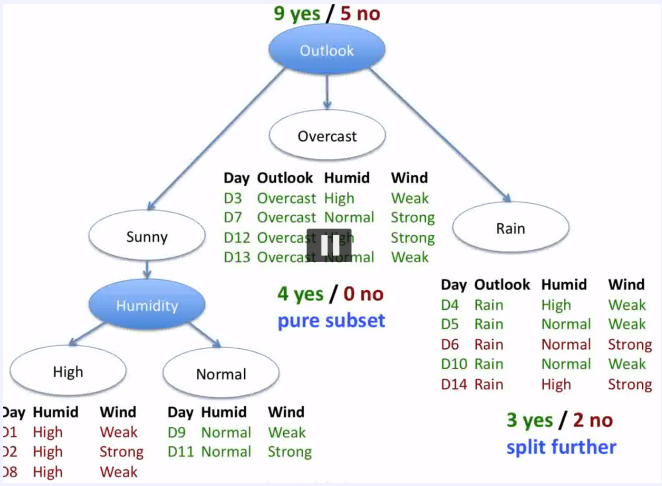
There is no need to fill in the gaps artificially because missing attribute data essentially has a probability of 1:



Decision Trees

Method

Decision trees split the data based on the label of each data point, in a way which maximizes the entropy of each split. This means we are splitting on attributes which have the most "discriminatory" power at each step



We classify each new sample by following the splits in the tree untill we hit a pure subset (or non-pure), and select the label based on the most frequent label in that leaf.

Quinlan's ID3 Algorithm

- A ¡- best attribute for splitting (**needs a heuristic**)
- Decision attribute for this node ¡- A
- For each value of A, create new child node
- Split training data at this node into child nodes
- For each child node/subset
  if subset is **pure**, stop
  otherwise repeat on this subnode

Purity measures

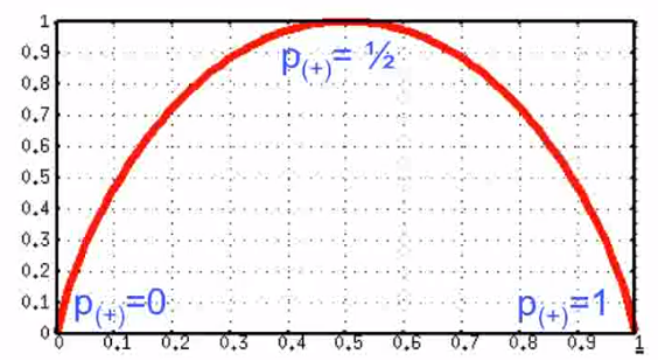
In order to pick the best attribute we need a purity measure
Even splits do not give us any information, we want ones biased towards the +'ve or -'ve labels
i.e. the purity for a pure set (4y/0n) is 100% but the purity for an impure set (3y/3n) is 50% (uncertain)

Entropy

$$H(S) = -p_+ \log_2 p_+ - p_- \log_2 p_- \tag{6}$$

where:

- S is the subset of training examples
- p's refer to the proportion of positive and negatives in the subset S



we interpret this measure as the number of bits of information needed to tell if a random item in S is a + or a -, we dont need any information at all if the subset is entirely composed of +'s or -'s, and all the information if it's evenly split

Information gain

once we have a measure of purity, we can establish the heuristic which will define which split will be the best:

$$Gain(S, A) = H(S) - \sum_{V \in Values(A)} \frac{|S_v|}{|S|} H(S_v) \tag{7}$$

where:

- **S** : subset of data points
- **A** : attribute we are splitting on
- **V** : possible values of attribute A
- S_v : subset of S with data points which have label v

This measures the "information" we gain by performing the split. Notice how we take a weighted average (weighted by the number of data points) of the entropy of each new subset formed in the split. If the entropy in the new subsets is on average higher than the entropy in the original subset, we have lost information (note this is not possible in this scenario) and if it's lower, we have indeed gained information. The split which maximises this quantity is the best.
Note: that picking the split on this criterion, implies that no attribute will be picked twice (since if we split on it once, all the subsets in nodes below will have the same value for this attribute)

Gain ratio

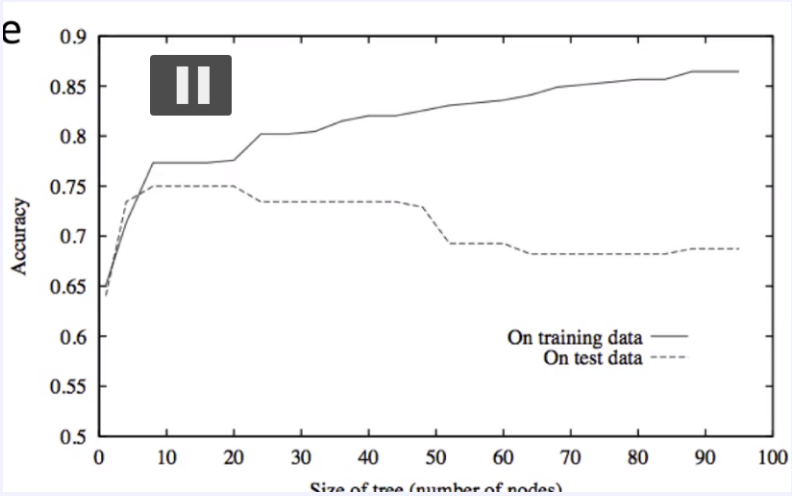
Using this heuristic will lead the algorithm to pick splits which lead to more subsets naturally (if allowed by shape of attributes and data) We can use the entropy of the split itself to counteract this instead:

$$\begin{aligned} SplitEntropy(S, A) &= -\sum_{V \in Values(A)} \frac{|S_v|}{|S|} log \frac{|S_v|}{|S|} \\ GainRatio(S, A) &= \frac{Gain(S, A)}{SplitEntropy(S, A)} \end{aligned} \tag{8}$$

not how the gain will be lower when we have too many or too few subsets, the split ratio will be the highest with higher information gains which result in 2 subsets that each take half of the data.

Overfitting

Overfitting happens when the model learns the "noise" in the training data, and becomes too specific.
Decision trees will always classify training examples perfectly if we make them big enough (because eventually we will split the data into singletons, and those instances will always be labelled as the correct instances)
This is an example of **overfitting**, we need to limit the size of the tree in order to counteract this, i.e. force some subsets to end up being non-pure in order to stop the model from becoming too specific to our training data.



Solutions

- Stop splitting when not statistically significant
- Grow the tree fully and then prune in a way which maximises the accuracy (using a validation set)

  - **Sub-tree replacement pruning**: pretend you remove a node and all its children, measure the performance on validation set. Remove node with highest performance gain, repeat untill removal is harmful to performance(greedy)

Numeric attributes

How can we apply this to numeric attributes? Split based on **thresholds**, threshold can be picked via some process. The easy solution is to take all the values for the attribute we are evaluating a split on, sort them and go through the split on each of those one-by-one.
Note: this will lead to one attribute possibly being split multiple times since we can always divide the values into smaller subsets

Multi-class classification

We can apply decision trees to multi-class classification by predicting the most frequent class in the subset we land on. The entropy calculation becomes:

$$H(S) = -\sum_{c} p_c \log_2 p_c \tag{9}$$

where p_c is the fraction of samples of class c in the subset

Regression

We can also apply this algorithm to regression by taking the average of the training examples in the subset we land on as the predicted output value. In which case instead of maximising gain we grow the tree by minimising variance in the subsets. (pick split which reduces variance in the output the most!) We can also replace the average by linear regression at the leaves.

Random decision forest

- Pick a random subset S_r of training examples
- grow a full tree T_r (without pruning)
- when splitting pick from $d << D$ random attributes (i.e. hide some attributes for some trees)
- compute gain based on S_r instead of full set
- repeat K times to make K trees

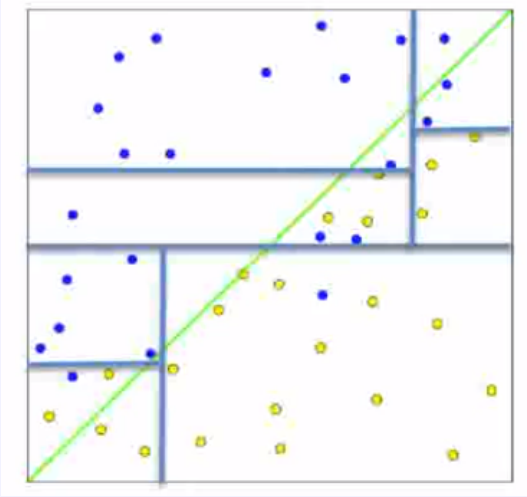
Given a new data point X we can then:

- classify X using each of the trees $T_1 \dots T_k$
- use majority vote: class predicted most often as the label!

State-of-the-art performance in many domains, very good stuff

Evaluation of decision trees

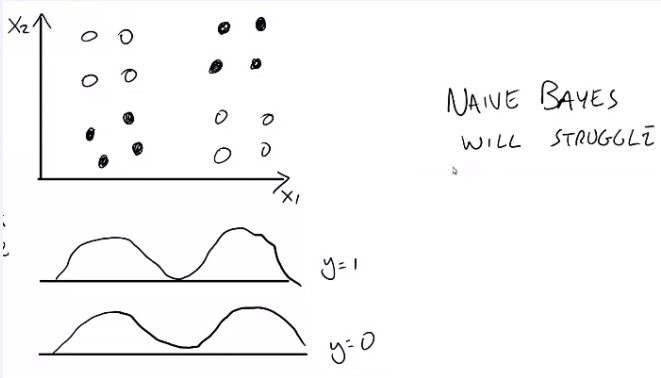
+ **interpretable** - humans can understand the underlying decisions! (ONLY ALGORITHM IN THIS COURSE TO HAVE THIS PROPERTY)
+ easilly handles irrelevant attributes (gain = 0)
+ can handle missing data
+ very compact: nodes << D after pruning
+ very fast at testing time O(depth)
- only axis-aligned splits are possible (because we split only taking into account one dimension at a step)



- greedy (may not find best tree globally)
- exponentially many possible trees

Why trees?

Can model non-linear, weirdly shaped data which does not really follow a nice probability distribution (i.e. **non-monotonic** data!)



Generalisation & Evaluation

Over & Under fitting

Overfitting

If we can find a model which makes more mistakes on the training data but fewer mistakes on unseen future data, our model has overfitted.
This can happen for many reasons:

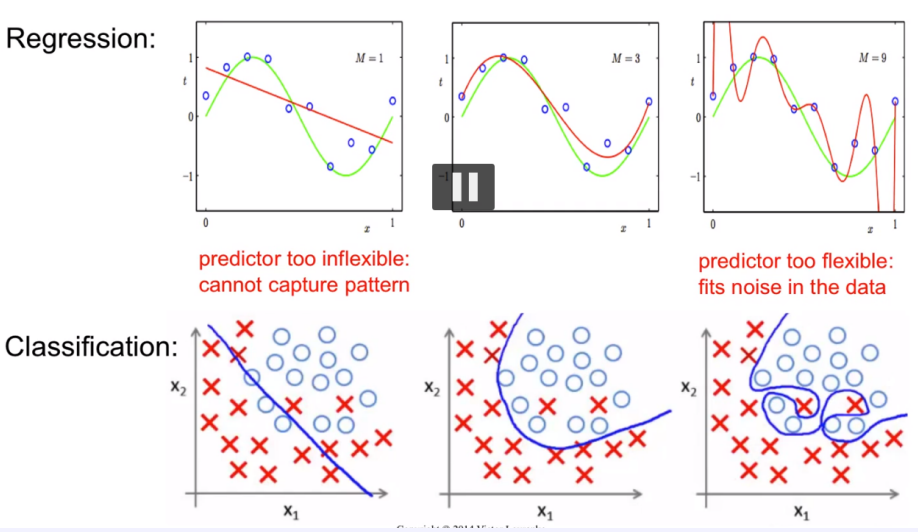
- Predictor is too "complex" (too flexible)
- It fits to the "noise" in the training data
- Captures patterns which will not-re appear in future data

Underfitting

If we can find another predictor with smaller training error and smaller error on future unseen data, our model has underfitted.
This can happen for many reasons:

- Predictor is too simplistic (too rigid)
- Not powerful enough to capture salient patterns in data

Examples



Flexible vs Inflexible

Every dataset requires a different level of flexibility. How much depends on the complexity of the task and the available data. We need a "knob" to vary the "flexibility" of our predictor.
Most learning algorithms have such knobs:

- Regression: order of the polynomial
- Naive Bayes: number of attributes, limits on model parameters
- Decision Trees: #nodes in the tree / prunning confidence
- kNN: number of nearest neighbours
- Support Vector Machines: kernel type, cost parameter

With a small amount of training data we require more rigit models (less chance for overfitting). For simpler tasks we also pick more rigid models. For complex tasks we want a more flexible/complex model

Training vs Generalization error

The training error can be defined as the average error over the training data (error here can be any error metric):

$$E_{train} = \frac{1}{n}\sum_{i=1}^{n} error(f_D(\mathbf{X_i}), y_i) \tag{10}$$

The generalization error however, is the general error on future data. Problem is we don't know what the future data will be, we do however know the possible range of input data.
Generalization error is defined as follows:

$$E_{gen} = \int error(f_D(\mathbf{X}), y)p(y, \mathbf{X})\ d\mathbf{x} \tag{11}$$

i.e. the sum over all possible input values of the error multiplied by the probability of seeing each of those input values given the class.
We can never compute this as we do not know $p(y, \mathbf{X})$, since that is what we're trying to model!
We can however estimate the generalization error, by treating the **testing** error as an estimate of it! (NOTE NOT THE TRAINING ERROR, NEVER USE THE TRAINING ERROR AS YOUR ACCURACY)

Estimating Generalization error

The testing error can be used to estimate generalization error:

$$E_{test} = \frac{1}{n}\sum_{i=1}^{n} error(f_D(\mathbf{X_i}), y_i) \tag{12}$$

If the testing set is an **unbiased** sample from the population of input data, i.e. unbiased sample of input values, then the following holds:

$$\lim_{n\to\infty} E_{test} = E_{gen} \tag{13}$$

I.e. as our sample size grows to infinity, our testing error converges onto the generalization error! How close are the two together ? this depends on n

Confidence interval

We can never actually gain the value for E_{gen} but we can estimate it using a previously unseen sample of the data, i.e. the test set. This estimate error E_{test} is going to be within some interval of E_{gen}. We can try to estimate how close to the real error we are using a confidence interval.
We can think of the error we measure as a sample from a distribution of all possible errors centered around the true error. From there we want to find an interval around the measured error describing our certainty of how close we are to the true error:

$$E_{test} \pm \Delta E_{gen} \tag{14}$$

$E_{gen} = E$ here is the probability that our system will misclassify a random instance (i.e. true error)
The confidence interval with p confidence is the range of values your error will be within p% of the time

$$\begin{aligned} CI &= E \pm \sigma \cdot \phi^{-1}\left(\frac{1-p}{2}\right) \\ \phi^{-1}(x) &= \sqrt{2} erf^{-1}(-p) \end{aligned} \tag{15}$$

With phi calculating the number of standard deviations of the normal distribution which covers p% of the future test sets.
we can now:

- take a random set of n instances, how many will be miss-classified ?
- flip E-biased coin n times, how many heads will we get ?
- this is the binomial distribution with mean = nE, variance = n E (1-E)
- $E_{future} = \#misclassified/n$, this can be approximated with a gaussian: with mean E, variance E(1-E)/n
- $\frac{2}{3}$ future tests will have error in $E \pm \sqrt{E(1-E)/n} = \sigma^2$
- Assuming E = 0.25 the confidence interval for future error will then be:
  - for n=100 examples, p=0.95:

$$CI = 0.25 \pm 1.96 \cdot \sigma = 0.25 \pm 0.08 \tag{16}$$

  - for n=1000 samples, p = 0.99

$$CI = 0.25 \pm 0.11 \tag{17}$$

  - for n=10000, p=0.95

$$CI = 0.25 \pm 0.008 \tag{18}$$

Note: the confidence interval varies roughly with the square root of n (if p is constant)

Data set splits

The test + training set split may not be enough, we also need a validation set to pick the right algorithm and "fine-tune" its parameters:

- Training set: construct basic classifier (Naive Bayes: count frequencies, DT: pick attributes to split on)
- Validation set: pick the algorithm + fine-tune settings
  - select best-performing algorithm
- Testing set: Estimate future error rate. Cannot be same as validation set, since the algorithm would overfit to the validation set after working out the parameters!
  - never report best of many runs, run once, or report results of every run

Holdout method

The most natural way to split up your data set is to pick a fraction of it as the training set, and the rest as the test/validation set, this is known as the **holdout** method

K-fold Cross-validation

Cross-validation is a technique for dealing with small data set sizes. Ideally we want to balance many goals:

- We want to estimate future error as accurately as possible - keep n_{train} as big as possible
- We want to learn the classifier as accurately as possible - keep n_{test} as big as possible

But both the training and test sets cannot overlap! We can however, split and alternate the sets in such a way to cover more ground:

- Split dadta into k sets
- test on each k in turn, while training on k-1 others
- average the result over k folds (why can we do that? because we never use one instance as both training/testing instance at the same time, also we are not averaging the performance of one model over number of runs, we build k different models!)

Note: while we call the sets test/train, we can use them for whatever purposes, the test set is actually likely going to be our validation set, and using cross-validation does not really set aside data for a full test set

Leave-one-out

Same thing as cross-validation ,but with K = n, i.e. each data point becomes a test set and the rest the training set.

+ best possible classifier is learned: n-1 training examples

- high computational cost: re-learn everything n times

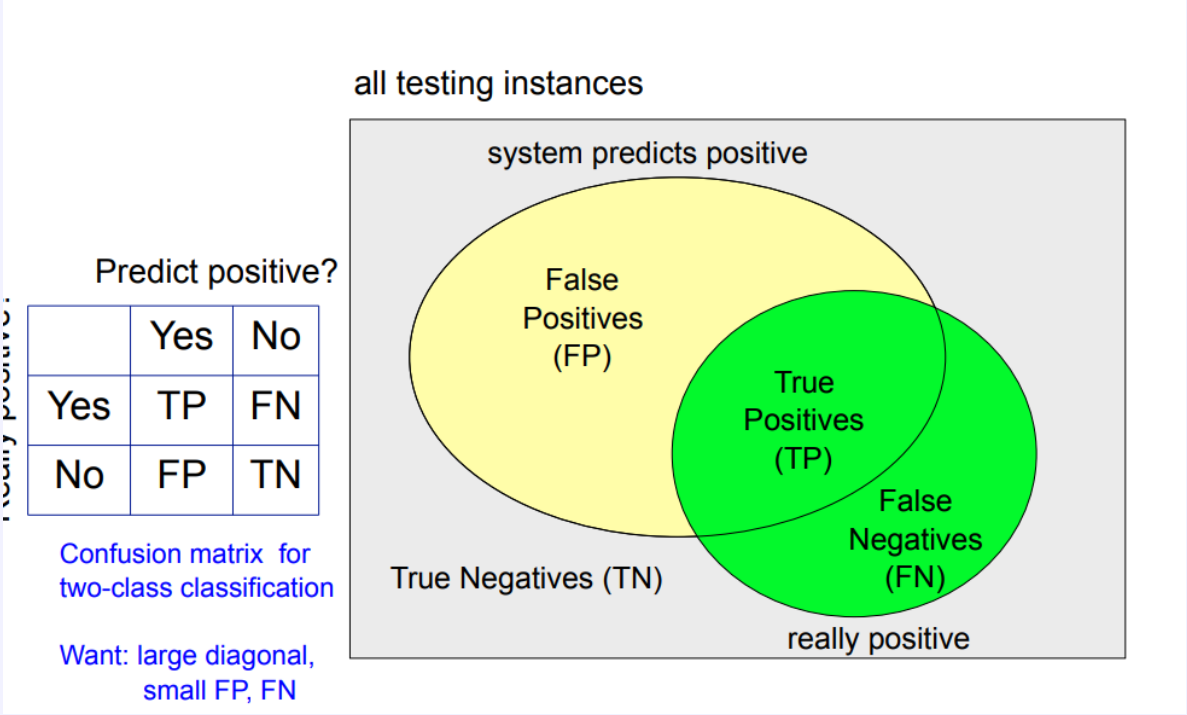
- classes are not balanced in training / testing sets

    - extreme example: say we have 2 equ-probable classes and **zero attributes**, this will lead to training set of n/2 samples of B and n/2 - 1 of A. Then the model must estimate that B is always the most likely, but the instance in the test set is of class A (same thing happens if B is in the test set, model is then biased towards A), so the model will always be wrong.

Stratification

All the above splitting methods disregard the class balance in the splits, this might be a problem.
The process of ensuring each split is always populated with equal proportion of class instances, is called **stratification**

Basics of classification measures

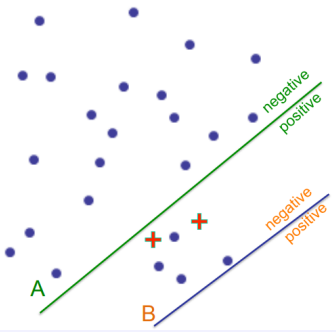


Accuracy/Classification Error

The percentage of correct classifications, or distance from real value to the predicted value in regression. This is not a great measure when the A priori are uneven, i.e. one class is more frequent

$$Acc = \frac{errors}{total} = \frac{FP + FN}{TP + TN + FP + FN} \tag{19}$$

- You're predicting Nobel prize (+) vs. not (•)
- Human would prefer classifier A.
- Accuracy will prefer classifier B (fewer errors)
- Accuracy poor metric here



Misses and false alarms

We can devise error metrics which aim to take into account the false positives and negatives more:

$$FalseAlarmRate = FalsePositiveRate = \frac{FP}{FP + TN}$$

% of negatives we misclassified as positive

$$Missrate = FalseNegativeRate = \frac{FN}{TP + FN}$$

% of positives we misclassified as negative

$$Recall = TruePositiveRate = \frac{TP}{TP + FN}$$

% of positives we classified correctly

$$Precision = \frac{TP}{TP + FP}$$

% of the things we predicted positive which were actually positive (dummy classifiers could be realy bad here)

(20)

We can use a combination of these to quantify the performance of a model, depending on what's more important and on the domain. i.e. it's trivial to get 100% recall or 0% false alarm ()

Utility and cost

We can combine these metrics into a single utility measure. If we know the cost of false positives, or negatives (i.e. how costly is it when our system misclassifies something, preventive measures, evacuation, or cost of recovery, reconstruction) We can also evaluate how costly our model is in our domain.

- Detection cost: weighted average of FP, FN rates, good for event detection (earthquakes)

$$Cost = C_{FP} * FP + C_{FN} * FN \tag{21}$$

- F-measure: harmonic mean of recall, precision, good for information retrieval, internet searches (takes out true negatives of the equation)

$$F1 = 2/(1/Recall + 1/Precision) \tag{22}$$

- Domain-specific measures:
  - e.g. observed profit/loss from +/- market prediction

Classification thresholds

Say two systems have the following performance:

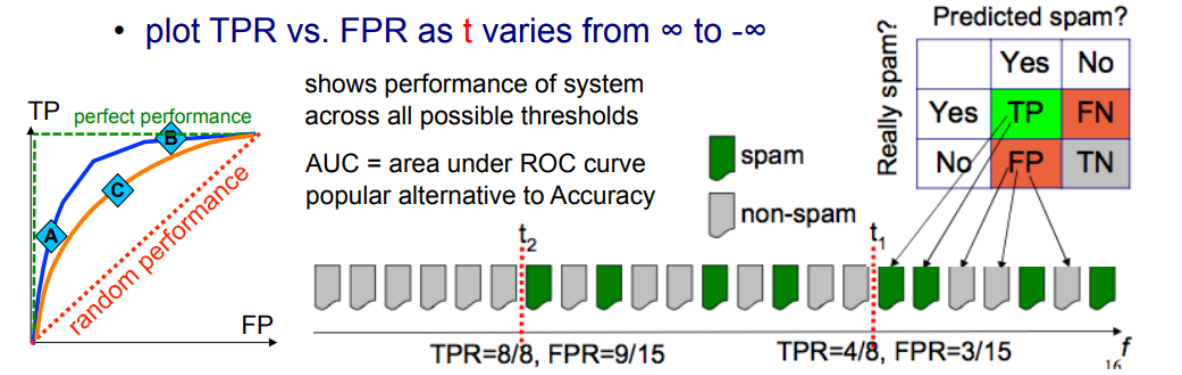
- A: True positive = 50%, False Positive = 20%
- A: True positive = 100%, False Positive = 60%

which is better ? (assume no-apriori utility)

- very misleading question, A and B could be **THE EXACT SAME SYSTEM** but operating at different **thresholds**

ROC curves

many algorithms computeu "confidence" c(x) i.e. the threshold above which they classify something as a positive. For naive bayes this is naturally 0.5
this threshold determines the error rates of the model. FP rate = $P(c(x) > t|ham)$, TP rate = $P(c(x) > t|spam)$
We can evaluate the performance of the model across the entire spectrum of t's using **Receiver Operating Characteristic** curves. Simply plot the TPR vs FPR as t varies from $\infty$to $-\infty$



The AUC is a good metric for comparing different classifiers which can operate over a variety of thresholds.

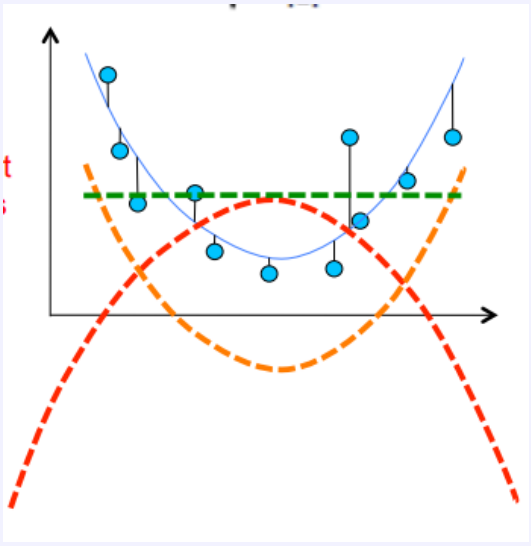
Evaluating regression

All the above methods deal with discrete, classifiers. How do we evaluate regression ? There are many error measures for regression:

Mean Squared Error

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i - y_i)^2 \tag{23}$$

popular, well understood, differentiable but sensitive to outliers, asingle prediction which is off by 10, is equivalent to 100 predictions off by 1. also sensitive to mean and scale



here orange predictor is not as bad, but will yield high error

Mean Absolute Error

$$\frac{1}{n}\sum_{i=1}^{n} |f(x_i - y_i)| \tag{24}$$

less sensitive to outliers, many small errors = one large error. Sensitive to mean and scale

Median Absolute Deviation

$$median(|f(x_i - y_i)|) \tag{25}$$

robust, completely ignores outliers. difficult to wor with - cannot take derivatives, sensitive to mean and scale

Correlation coefficient

$$\frac{n\sum_{i=1}^{n} f(x_i - \mu_f)(y_i - \mu_y)}{\sqrt{n\sum_{i=1}^{n} f(x_i - \mu_f)^2 (y_i - \mu_y)^2}} \tag{26}$$

Basically ignores the "position" of your prediction function and likes predictors of similar shape to the data (without considering mean and scale) *insensitive to mean & scale, good for ranking tasks. Intuition: is your output small as target is small and vice-versa?*

Linear regression

Linear model

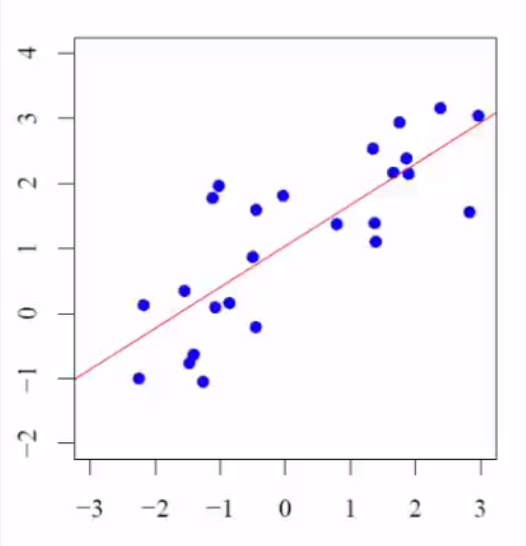
The idea is to assume the output value is a linear combination of the attributes, i.e.:

$$\begin{aligned} y = f(\mathbf{x}) &= w_0 + w_1 x_1 + \ldots + w_D x_D = \phi(\mathbf{x})\mathbf{w} \\ \phi(x) &= (1, x_0, x_1, \ldots, x_D) \end{aligned} \tag{27}$$

where:

- $\mathbf{w}$ are the weights, or model parameters (think gradient of the line)
- $\mathbf{x}$ is the observation
- $\phi(x)$ is just a generalization of x which appends a 1 to the beginning (and performs transformations on the input data)

Once we have the weights vector, we can predict new outputs by simply following the line defined by our model:



Note: if $\mathbf{x}$ *is the matrix of all input data (i.e. each row is an input data point),* $\phi(\mathbf{x})$ *becomes a matrix and* $\mathbf{y}$ *becomes a vector or predictions for each data point*

Calculating w - least squares

We need to find the parameters $\mathbf{w}$ given the **design matrix** ϕ (where each training instance is a row). Many methods, simplest is to minimise loss function:

$$O(\mathbf{w}) = \sum_{i=1}^{n} (y_i - \mathbf{w}^t x_i)^2 \tag{28}$$

i.e. we are finding the weights which give us a model with *the least squared error* - or minimises sum of **residuals** squared.
This problem is exactly the same as the problem of finding a solution to tall matrices of equations $Ax = b$ which minimizes $|Ax - b|^2$. The standard way of dealing with tall matrices is using the **pseudo-inverse**:

$$\mathbf{w} = (\phi^T \phi)^{-1} \phi^T \mathbf{y} \tag{29}$$

Probabilistic intepretation of O(w)

This method can be interpreted probabisticially by assuming: $y = \mathbf{w}^T \mathbf{x} + e$
where e is just some arbitrary gaussian noise $N(0, \sigma^2)$. This means, we are assuming that our data is scattered around the hyperplane of best fit according to some gaussian noise. This implies that $y|\mathbf{x}_i \approx N(\mathbf{w}^T \mathbf{x}_i, \sigma^2)$ So minimising $O(\mathbf{w})$ is equivalent to maximising the likelihood of this model! (probability output value is drawn from this gaussian)
Then $\mathbf{w}^T \mathbf{x}$ can be seen as the expectation $E(y|\mathbf{X})$ and the squared residuals give us the estimation of the "noise" model:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \tag{30}$$

Linear regression with multiple outputs

what if for each piece of input data we want to predict more than one value ? Introduce a $\mathbf{w_y}$ for each y required and perform **multiple regression**, i.e. a separate regression for each target.

Non-linear data

We can expand linear regression to work with non-linear data, by transforming it first!

$$\phi = \begin{bmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \dots & \phi_3(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \dots & \phi_3(\mathbf{x}_2) \\ \vdots & \vdots & \dots & \vdots \\ \phi_1(\mathbf{x}_d) & \phi_2(\mathbf{x}_d) & \dots & \phi_d(\mathbf{x}_2) \end{bmatrix} \tag{31}$$

The solution to w stays the same and this way we can establish linear relationships with non-linear transforms of the data!

Polynomial regression

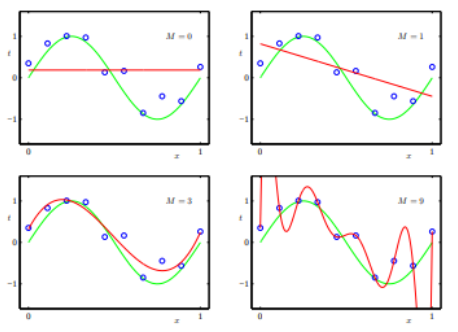
an example of this is polynomial regression of the form (with 1D data):

$$\begin{aligned} \phi_1(x) &= x \\ \phi_2(x) &= x^2 \\ \phi &= (1, x, x^2) \end{aligned} \tag{32}$$

this would yield solutions of the form:

$$y = w_0 + w_1 x + w_2 x^2 \tag{33}$$

I.e. the model would find a hyperplane in higher dimensional space which actually expresses the polynomial form we're looking for!



Radial basis functions

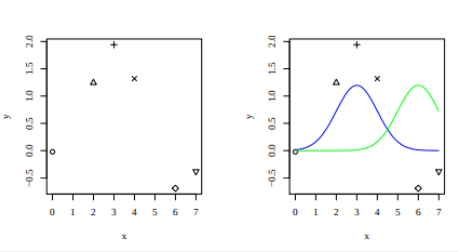
Another common transformation is **RBF** defined as:

$$\phi_i(\mathbf{x}) = \exp\left(\frac{-\frac{1}{2}\left|\mathbf{x} - \mathbf{c}_i\right|^2}{\alpha^2}\right) \tag{34}$$

where:

- α is the "width" of the basis function, akin to variance in a gaussian distribution
- $\mathbf{c_i}$ is the "center" of the basis function, akin to the mean in a gaussian distribution

Notice how this is equivalent to a gaussian distribution, just without the normalizer term (i.e. this does not sum to 1)
Example with 1D data and 2 radial basis:



hard to imagine, but this way we express the output as a linear combination of distances to each centre, and this can give us a nice linear relationship.
These are really good, but with more dimensions you'll need way too many of them and that probably means overfitting, also hard to learn the rbf parameters

one-hot-encoding

How do we translate categorical features to work with linear regression ? can we give a different numeric value to each category like so?:

$$x_1 = \begin{cases} 1 & \text{if intel} \\ 2 & \text{if AMD} \\ 3 & \text{if Apple} \\ 4 & \text{if Motorola} \end{cases} \tag{35}$$

No! HELL NO. EW!
Think about it, say we have only this attribute and no bias:

$$y = x_1 \cdot w_1 \tag{36}$$

maybe intel produces twice as good processors as AMD (based). We cannot express that in this form, we only get one weight for the attribute, so changing w_1 equally changes the **contribution** of each manufacturer to the output (clock speed or whatever).
We can however split up each value of this attribute into its own attribute:

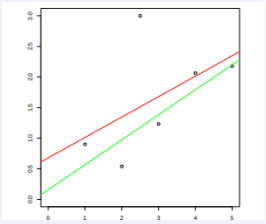
$$\begin{aligned} &x_1 = 1 \text{ if intel or } 0 \text{ otherwise} \\ &x_2 = 1 \text{ if AMD or } 0 \text{ otherwise} \\ &\vdots \end{aligned} \tag{37}$$

We can then express the superiority of intel with weights $w_1 = 1$,$w_2 = 0.5$ (exact values, scale etc.. irrelevant rn.)like so:

$$y = x_1 + x_2 \cdot 0.5 + \dots \tag{38}$$

Evaluation of Linear regression

+ very simple yet powerful
+ **invariant to scaling of attributes** (no need for normalization)
- sensitive to outliers



Logistic regression

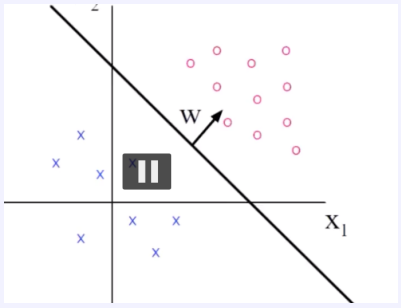
Binary classification

While the machine learning mafia would like you to believe this algorithm is used for regression, this is shockingly not the case, we use it to perform binary classificatin like so:

$$y = f(\mathbf{x}) = \begin{cases} 1 & \text{if} \quad \mathbf{w}^T\mathbf{x} + w_0 > 0 \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

where:

- $\mathbf{w}$ is the weight vector perpendicular to the decision boundary (pointing to the "side" of class 1)
- w_0 is the bias term (extracted from w for convenience)



I.e. we model the decision boundary instead of the distributions of the classes (**discriminative** model) Decision boundary (hyperplane) is defined as $\mathbf{w}^T\mathbf{x} + w_0 = 0$ (think with $w_0 = 0$ dot product of x with w must be zero if x is on the line, w_0 offsets this linearly)

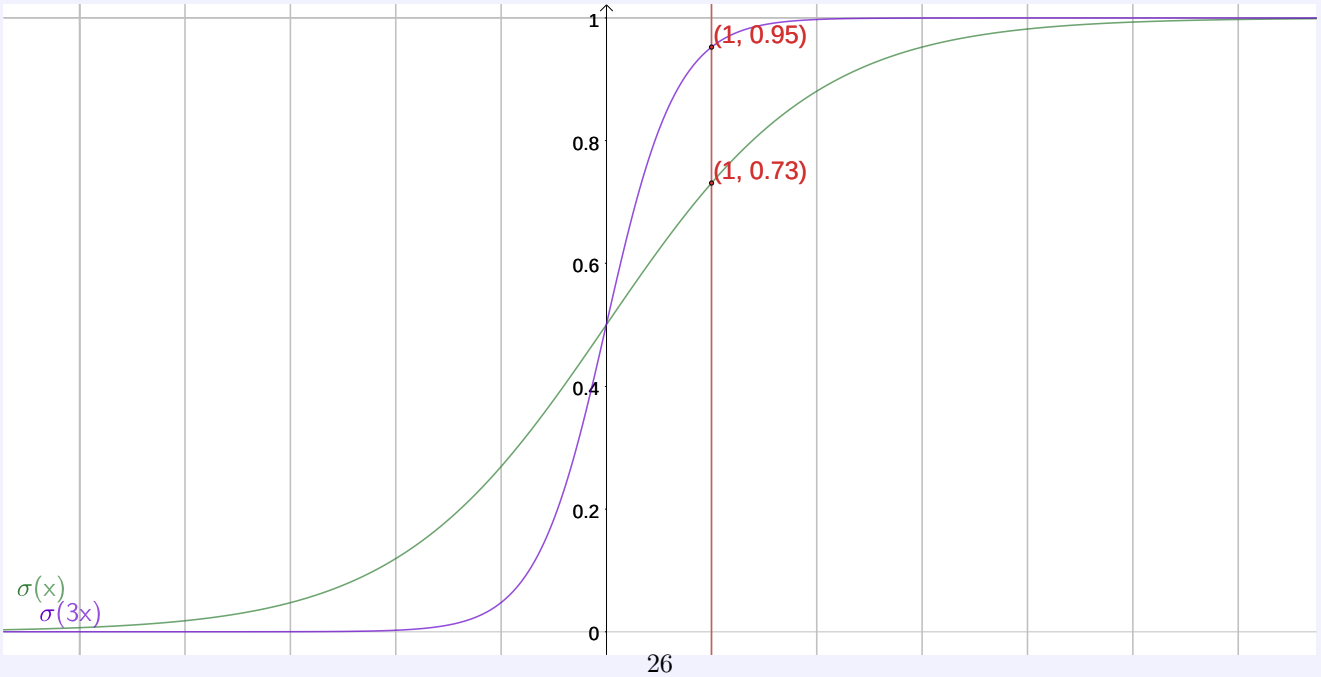
Probabilistic version

We can do a bit better, and give ourselves the benefit of a probability value so we can say how confident we are in our prediction, by "squashing" the distances from the hyperplane so that they form probabilities: $P(y = 1|\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ does not work, because the value is unbounded and breaks probability axioms, instead we use a sigmoid:

$$\begin{aligned} P(y = 1|\mathbf{x}) &= f(\mathbf{w}^T\mathbf{x}) \\ P(y = 0|\mathbf{x}) &= 1 - P(y = 1|\mathbf{x}) \\ f(z) = \sigma(z) &= \frac{1}{1 + \exp(-z)} \end{aligned} \tag{40}$$

The sigmoid function maps extreme distance values to the range [0,1]:
since $\sigma(z) = 0.5$ when $z = 0$, the **decision boundary** is given by $\mathbf{w}^T\mathbf{x} = 0$
The scale of $\mathbf{w}$ affects the shape of the sigmoid, larger $\mathbf{w}$ inflate the values, scaling the sigmoid on the x axis by the scale factor effectively "squashing" it.

- larger $\mathbf{w} \rightarrow$ probabilties increased (mostly near decision boundary)
- smaller $\mathbf{w} \rightarrow$ probabilities decreased (mostly near decision boundary)



Calculating w

To find the model parameters we derive probabilistic expression for likelihood and maximise it.
Let dataset $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ Then the likelihood of our data (probability of seeing this data with w being the weight vector) is:

$$\begin{aligned} p(D|\mathbf{w}) &= \prod_{i=1}^{n} p(y = y_i|\mathbf{x}_i) \\ &= \prod_{i=1}^{n} p(y = 1|\mathbf{x}_i)^{y_i}(1 - p(y = 1|\mathbf{x}_i))^{1-y_i} \quad \#\text{split by cases} \\ L(\mathbf{w}) &= \sum_{i=1}^{n} y_i \log p(y = 1|\mathbf{x}_i) + (1 - y_i) \log(1 - p(y = 1|\mathbf{x}_i)) \quad \#\text{log likelihood} \\ L(\mathbf{w}) &= \sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \end{aligned} \tag{41}$$

Now to maximise we find gradient, and find maxima, cannot do that **analytically** however, need to use a **numerical** technique. The gradient (worked out by simple application of multivariate calculus) is:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_d} \end{bmatrix} \qquad \frac{\partial L}{\partial w_j} = \sum_{i=1}^{n} (y_i - \sigma(\mathbf{w}, \mathbf{x}^T \mathbf{x}_i)) x_{ij} \tag{42}$$

The gradient points in direction of steepest ascent, i.e. direction of change to w which will increase likelihood

Gradient descent

We can apply **gradient descent** to find the local maximum - which turns out to be the global maximum because the gradient is a **convex** function!

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \nabla L(\mathbf{w_t}, \mathbf{x}) \tag{43}$$

with α here standing for the learning parameter which balances the accuracy vs speed of the descent

Multi class classification

To apply this algorithm to a multi-class classification problem:

- Introduce a weight vector $\mathbf{w}_k$ for each class k, which does k vs not-k classification.
- Apply softmax to pick the most probable class:

$$p(y = k|\mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{j=1}^{C} \exp(\mathbf{w}_j^T \mathbf{x})} \tag{44}$$

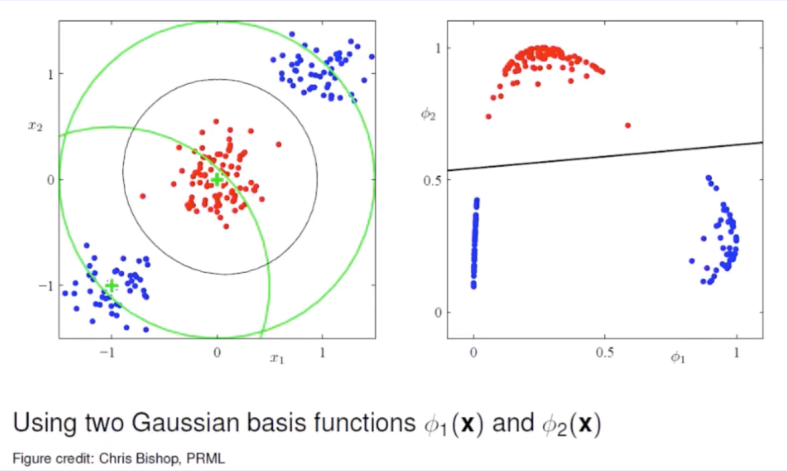
Note that: $0 \leq p(y = k|\mathbf{x}) \leq 1$ and $\sum_{j=1}^{C} p(y = j|\mathbf{x}) = 1$, i.e. this is another probability distribution but over all the classes. **softmax** is a natural generalization of the sigmoid function to more than 2 classes, and will "split" the probability between classes. We could simply take the sigmoid outputs for each class if our classes are not **mutually exclusive** (for example, if classes are: {has leg, has arm})

Linear Separability

Logical regresion, divides the space by a hyper plane and therefore can only perform well on problems which are linearly separable (classes can be divided by a plane). i.e. problems where we can find a $\mathbf{w}$ such that:

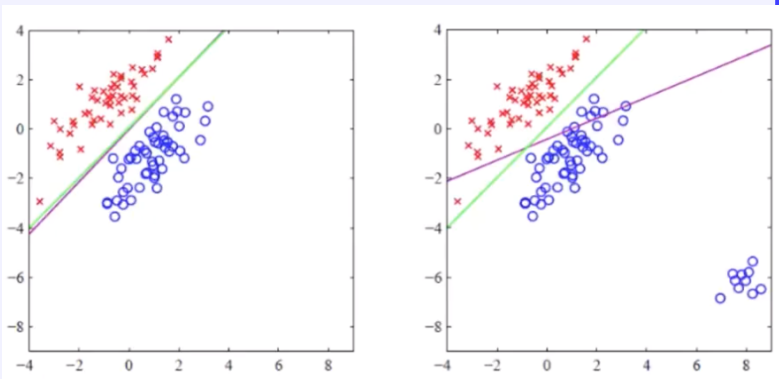
$$\begin{aligned} \mathbf{w}^T\mathbf{x} + w_0 &> 0 \quad \text{for all postitive classes} \\ \mathbf{w}^T\mathbf{x} + w_0 &\leq 0 \quad \text{for all negative classes} \end{aligned} \tag{45}$$

However, we can apply **non-linear** transformations to the input data (similarly as in linear regression), to make the data **linearly separable**.



Evaluation of logical regression

+ **discriminative advantage**: avoids spending time on generating probability model, models boundary directly (models $p(y|\mathbf{x})$ directly vs $p(\mathbf{x}|y)$)

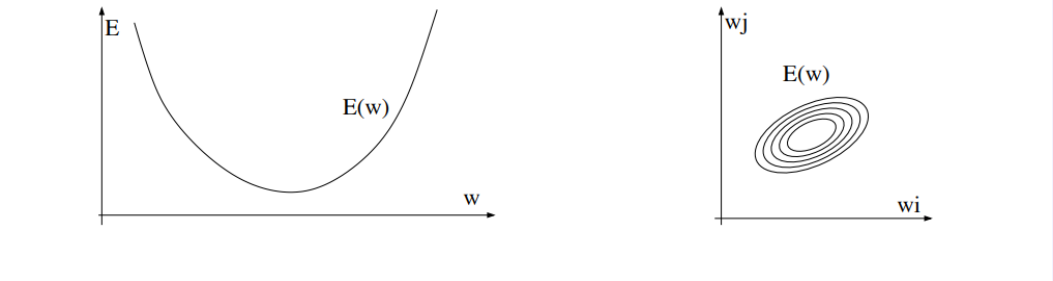


+ handles outliers better than linear regression
+ less assumptions about data (doesn't assume any distribution or underlying model) apart from separability
+ handles correlation relatively well
- doesn't handle missing data as gracefuly as NB (generative advantage)

Optimisation & Regularisation

Error function

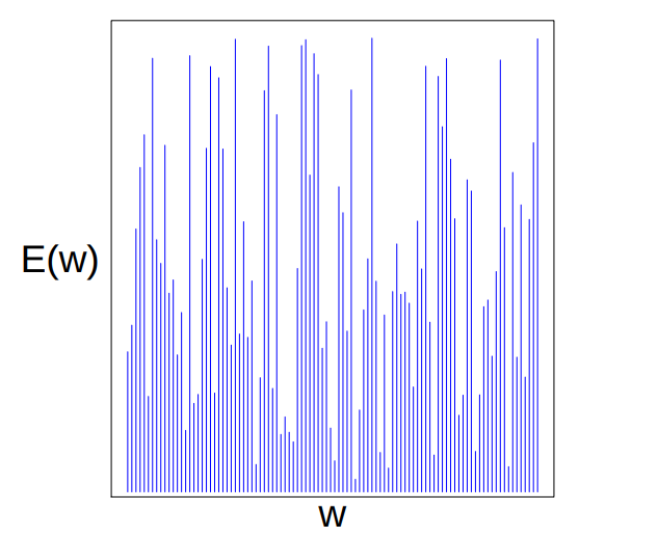
We can think of optimising the weight vector in terms of the error function defined against the weights: $E(\mathbf{w})$ which defines a surface:



Learning = descending this curve

Role of smoothness

If E is completely unconstrained, minimization is impossible, best we could do is search through all possible values of $\mathbf{w}$



Role of derivatives

The gradient of the error function tells us the direction we need to push $\mathbf{w}$ in to increase error:

$$\frac{\partial E}{\partial \mathbf{w}} = \nabla E = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{pmatrix} \tag{46}$$

Multi-variate calculus is cool, think of each partial derivative as "wiggling" that component of w and seeing how it affects the function, each direction of wiggling is independent when it comes to the gradient!

Numerical optimization

Numerical optimization tries to solve the general problem:

$$\min_{\mathbf{w}} E(\mathbf{w}) \tag{47}$$

Most commonly this is dune using the gradient (sometimes using more information, like higher order derivatives)
Most basic optimization algorithm is:

1. Initialize $\mathbf{w}$
2. while $E(\mathbf{w})$ too high, calculate gradient $\mathbf{g} = \nabla E$
3. pick direction d based on $\mathbf{g}$,$\mathbf{w}$,$E(\mathbf{w})$
4. set $\mathbf{w} = w - n\mathbf{d}$. Repeat from 2.

note n is the learning rate or step size.

Choice of direction

Simplest choise for $\mathbf{d}$ is the current gradient ∇E (because we are moving in direction of -d)

Gradient descent

Numerical optimizatio where at each step we choose $\mathbf{d} = \nabla E$.
If the **step size** is too small, the algorithm will be slow, and if too large, it won't be stable

Bold Driver gradient descent

Simple heuristic for choosing n which you can use if desperate:

- Perform gradient descent but: we repeat while n is greater than zero
- At each step if the error has increased from the previous step, we set $n = n/2$
- If the error has not increased, we increase n (say by factor of 1.01)

Batch vs Online learning

Batch

Batch learning uses all patterns in the training set, and updates the weights after

$$\nabla E = \sum_i \nabla E_i \tag{48}$$

Note E_i signifies the error due to i'th sample.
Batch methods are easier to analyze and more powerful.

Online learning

Online learning uses a single instance at each iteration and updates weights after each

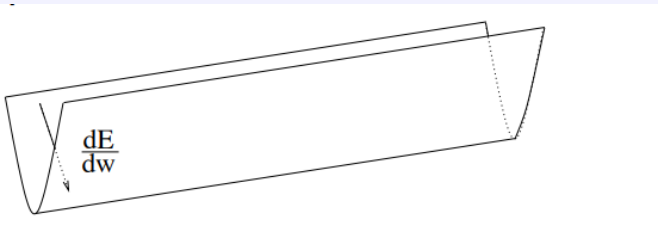
$$\nabla E = \nabla E_i \tag{49}$$

Online methods are more feasible for huge or continually growing datasets. Online learning allows the possibility of jumping over local optima ocasionally

Problems with gradient descent

Shallow valleys

gradient descent can be made very slow if the shape of the error function looks like a valley: The algorithm will travel very slowly once it hits the valley.



One solution is to use momentum in place of the gradient:

$$\mathbf{d}_t = \beta \mathbf{d}_{t-1} + (1-\beta) n \nabla E(\mathbf{w}_t) \tag{50}$$

Here β varies how much momentum we inherit from the previous iteration. But this increases the complexity of the algorithm, and you have to find both optimal values for n and β

Curved error surfaces

The gradient always points in the direction of steepest ascent, but this might not actually be any local maximum due to curvature. This means gradient descent is not always very fast.
We can measure local curvature using the hessian matrix:

$$H_{ij} = \frac{\partial^2 E}{\partial w_i w_j} \tag{51}$$

Local minima

While following the gradient guarantees that we hit a local maximum, this maximum is not necessarily global. Meaning that we can get stuck in a solution which is sub-optimal. If the error function is not **convex** we are not guaranteed to find the global maximum.
The best you can do is run the algorithm from different points and pick best run.

Regularization

The process of limiting the complexity of parameters or extreme values. For example in linear regression we can add a secondary objective to the optimization problem i.e. $\mathbf{w}$ being as small as possible (perhaps via lagrange multipliers)
Ridge regression is a good example of a regularized model - here we modify linear regression and try to introduce a secondary goal, i.e. minimizing the complexity of the model $\mathbf{w}$

Support Vector Machines

Support Vector Machine

Support vector machines are **discriminative** predictors, they find the decision boundary by maximising the size of the margin (distance between boundary and closest training sample)
Classification is done with:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + w_0) = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i (\mathbf{x}_i^T \mathbf{x}) + w_0) \tag{52}$$

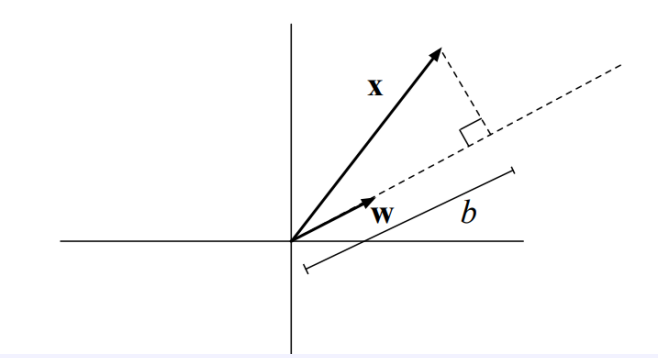
where:

- α_i is non-zero only for support vectors
- y_i is the label (1 for points on the side of hyperplane to which $\mathbf{w}$ points, and -1 for the other side)
- $\mathbf{w}$ is the weight vector pointing towards label 1
- $\mathbf{x}$ is the new data point
- $\mathbf{x}_i$ are the training data points

if you look at this you'll see that essentially, you are summing "dot-similarities" to the support vectors on one side as positives and the "dot-similiarities" to the support vectors on the other sides as negatives, meaning that if your data is more "similar" to the positive side the output will be positive and negative otherwise.

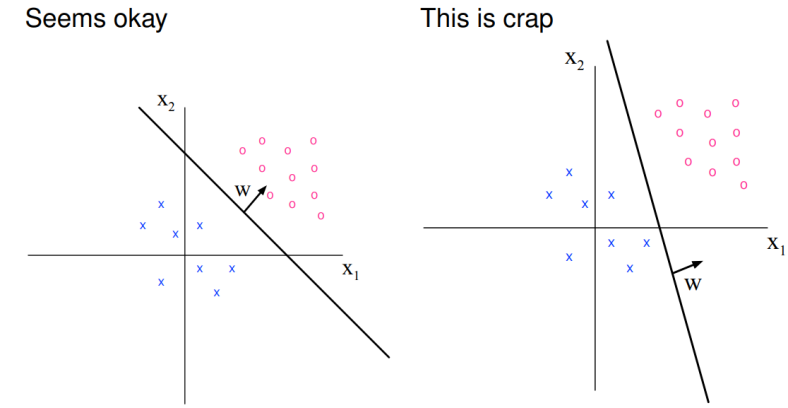
Scalar projection on a unit vector

If $\mathbf{w}^T$ is a unit vector then $\mathbf{w}^T\mathbf{x}$ is the length of the projection of $\mathbf{x}$ onto $\mathbf{w}$



Separating hyperplane

The decision boundary between classes is called the separating hyperplane, for SVM's the labels on the opposite sides of a hyperplane are (1,-1) as opposed to the standard (1,0) for mathematical convenience.
If $\mathbf{w}^T$ is the vector perpendicular to the hyperplane then the decision boundary is defined with $\mathbf{w}^T\mathbf{x} + w_0 = 0$



Bad decision boundaries should be as general as possible, i.e. have **highest margin**.

Margin

the **margin** is the distance between the decision boundary and the closest training point
this distance can be found geometrically via:

$$\min_i \frac{1}{||\mathbf{w}||}|\mathbf{w}^T\mathbf{x}_i + w_0| \tag{53}$$

Because an infinite amount of $(\mathbf{w}^T, w_0)$'s describe the same hyperplane (all vectors parallel to $\mathbf{w}^T$ have same direction), and since we can always multiply $(\mathbf{w}^T, w_0)$ by any scalar, we impose an additional constraint:

$$\min_i |\mathbf{w}^T\mathbf{x}_i + w_0| = 1 \tag{54}$$

which means the margin is always defined as (assuming the constraint is upheld):

$$m = \frac{1}{||\mathbf{w}||} \tag{55}$$

Finding maximum margin

We can define the optimisation problem as :

$$\begin{aligned} &\max_{\mathbf{w}} \frac{1}{||\mathbf{w}||} \\ &\text{subject to:} \\ &y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 \quad \text{for all } i \end{aligned} \tag{56}$$

Which is really just saying, maximise the margin but keep my boy $\mathbf{w}$ nice and unique, and them data points on the right sides of the hyperplane.
This problem is equivalent to:

$$\begin{aligned} &\min_{\mathbf{w}} ||\mathbf{w}||^2 \\ &\text{subject to:} \\ &y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 \quad \text{for all } i \end{aligned} \tag{57}$$

Of course the spooky maths required (quadratic programming, lagrangian multipliers or something) is not covered so we are not shown how to solve this, but using proof by professor, the optimal parameters are:

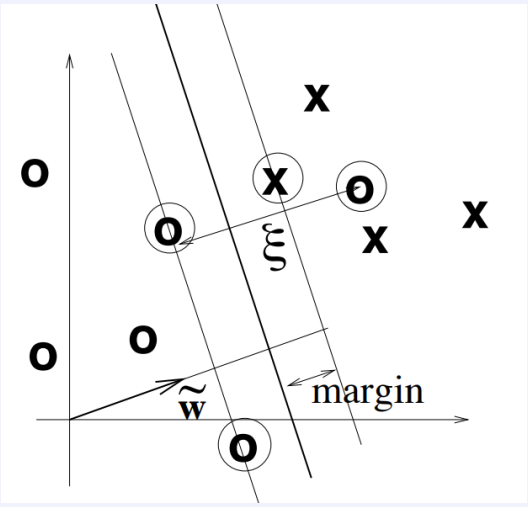
$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \tag{58}$$

where:

- α_i which states which vectors are support vectors - it is non-zero only for support vectors (data points on the margin)
- $y_i \in -1, 1$ is the label (or side of hyperplane)

Non-separable data sets

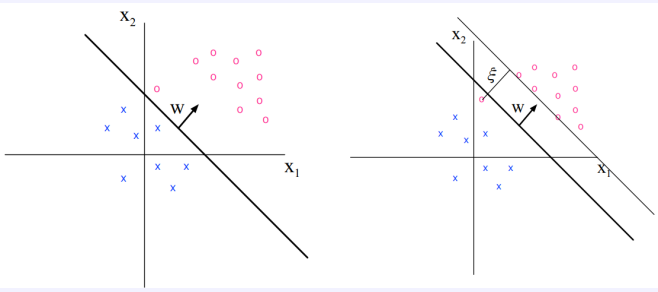
The above work for linearly separable data sets, otherwise we have to allow the boundary to mis-classify some points, we use **slack** for that (constraint which stops the optimisation from miclassifying all points, since that's allowed):



Optimisation problem becomes, minimizing:

$$
\begin{aligned}
&||\mathbf{w}||^2 + C(\sum_{i=1}^{n} \epsilon_i)^k \\
&\quad \text{subject to:} \\
&\mathbf{w}^T\mathbf{x}_i + w_0 \geq 1 - \epsilon_i \quad \text{for } y_i = +1 \\
&\mathbf{w}^T\mathbf{x}_i + w_0 \leq -1 + \epsilon_i \quad \text{for } y_i = -1
\end{aligned}
\tag{59}
$$

k is usually set to 1 and C is a trade-off parameter, larger C's allow less error (more penalty for errors) The solution to this is exactly the same as previously, but this time more vectors are support vectors (even ones for which $\epsilon \neq 0$).
Notice how this is very similiar to ridge regression (minimizing $||y_i - \mathbf{x}^T||^2 + \lambda||\mathbf{w}||^2$) The penalty term measures how well we fit the data, and the w term penalizes weight vectors with large norm. So C can be seen as a regularization term since it increases or decreases the complexity of the model just like λ.



We can solve this by rewriting this as the following:

$$
\min_{\mathbf{w}\in\mathbb{R}^d} ||\mathbf{w}||^2 + C\sum_i \max(0, 1 - y_i f(\mathbf{x_i})) \tag{60}
$$

where the bit in the summation is called the **hinge loss**

Non-linear SVM's

SVM's are normally linear, but can be made non-linear by applying basis transformations on the input vectors as usual.

Kernel trick

Normally when applying a non-linear transformation on x we transform x to the new space and then perform calculations on the new data. However if we are only interested in some sort of "similarity" of vectors in new spaces, we can apply the kernel trick:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \tag{61}$$

i.e. we can take a shortcut and instead of transforming and then taking the dot product, we can take the dot product straight away!
Example:

$$\begin{gathered} \phi(\mathbf{x}) = \begin{bmatrix} \mathbf{x}_1^2 \\ \sqrt{\mathbf{x}_1 \mathbf{x}_2} \\ \mathbf{x}_2^2 \end{bmatrix} \\ k(\mathbf{x_1}, \mathbf{x_2}) = (\mathbf{x}_1^T, \mathbf{x}_2)^2 \end{gathered} \tag{62}$$

here the kernel computes the dot product of 2D vectors which are to be transformed to the above 3D space, without actually transforming them!

Distance

we can define a distance measure using the dot product in any space which is another benefit:

$$d(\mathbf{x}_1, \mathbf{x}_2) = k(\mathbf{x}_1, \mathbf{x}_1) - 2kk(\mathbf{x}_1, \mathbf{x}_2) + k(\mathbf{x}_2, \mathbf{x}_2) \tag{63}$$

Infinite spaces

the kernel:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp - \frac{||\mathbf{x_i} - \mathbf{x_j}||^\mathbf{2}}{\alpha^2} \tag{64}$$

actually corresponds to the dot product in an **infinite dimensional** space! Ain't that trippy.

SVM Evaluation

+ Great results in many problems
+ Hard to overfit (especially in high dimensional spaces) i.e. better for generalization than for example logistic regression
+ Solution is sparse, fast to calculate

Ethics

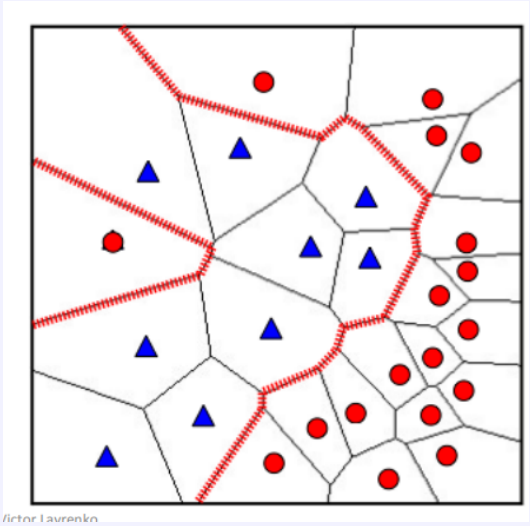
Just be a good boi

Nearest Neighbours

Nearest neighbour classification

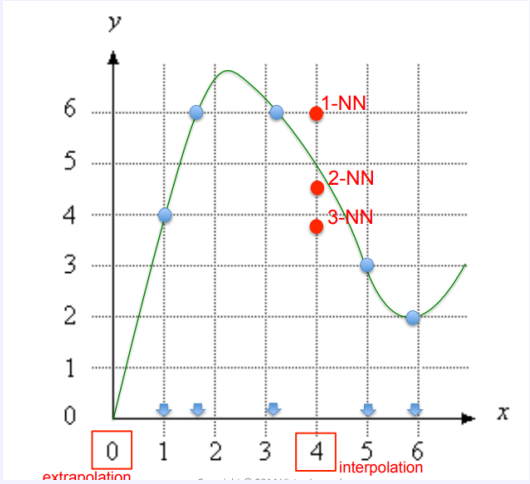
- Compute distance D of new instance **x** to every training example
- select k closest instances and their labels
- output the label/class most frequent in those labels

That's it. Makes real complex boundaries for such simple algorithm (voronoi tesselation):



Regression

You want numbers ? have numbers, take the average of the k neighbour labels instead of their labels.
Works worse for **extrapolation**



Choosing k

Value of k is strongly coupled to performance of this algorithm. if k = n, the output is simply the mean of the dataset. if k is too small we get unstable decision boundaries and overfit (+ small changes in training set = large changes in classification).
There isn't really a trick to picking k, just set aside a validation set and pick k which works best on it.
It is CRITICAL that the validation set is different from the training set, or otherwise k=1 is the most optimal always, since then there is always one example near each instance with the correct label

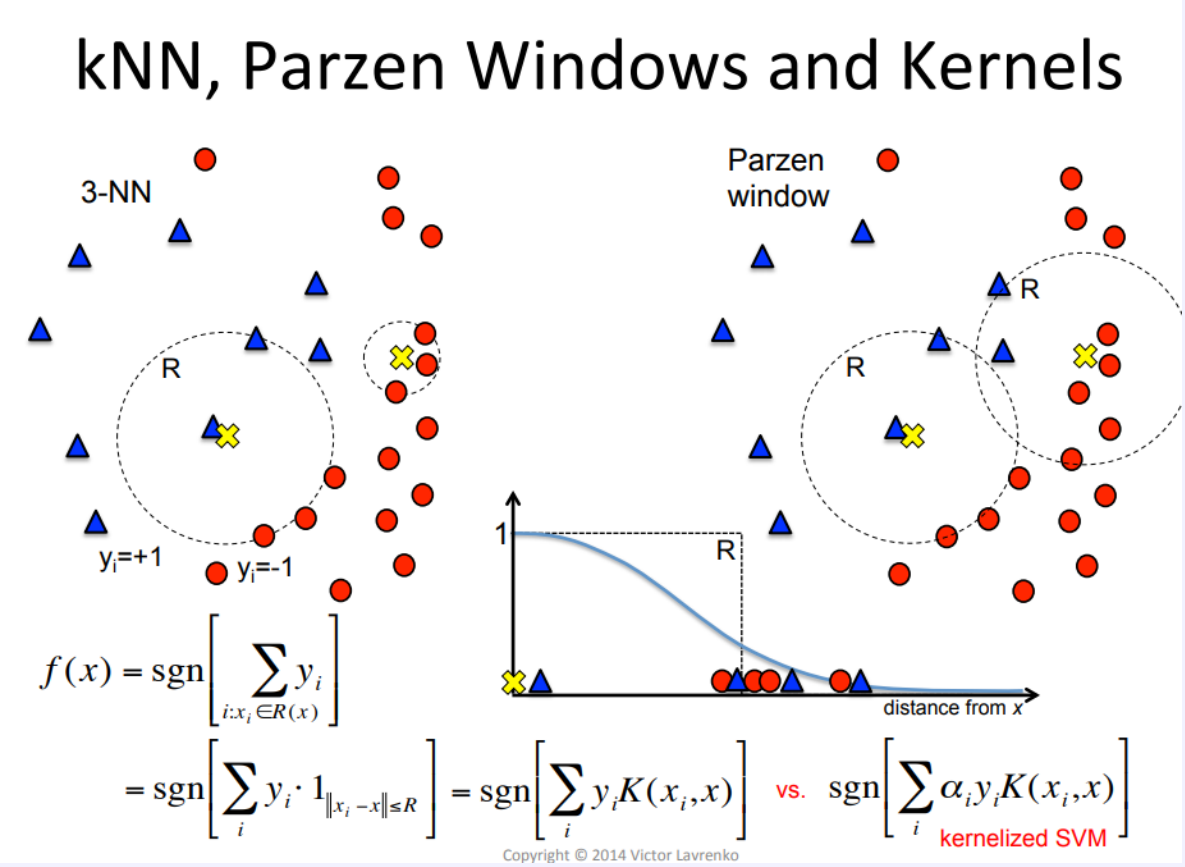
Ties

Sometimes we have equal number of positive/negative neighbours. Many ways we can handle this:

- use an odd K (doesn't solve multi-class classification)
- choose neighbour label at random
- use the nearest neigbhour
- pick class in neighbours with greatest prior

Parzen windows and kernels

Alternative is to fix a region of space around each new sample and pick most common class/value there



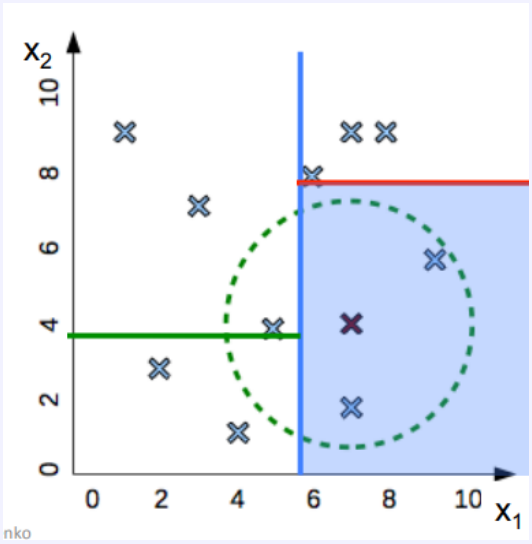
note how this is very blocky and can introduce a lot of **variance** to the model, with small changes in R we can affect the classifications drastically. We can smooth this out by using a smoother function which weights the neighbour label's by their distance. The kernelised version is extremely similar to SVM's but here we use all data points to make the decision whereas the SVM uses the support vectors only.

Optimising kNN

There are some ways which can speed up the kNN algorithm

K-D trees

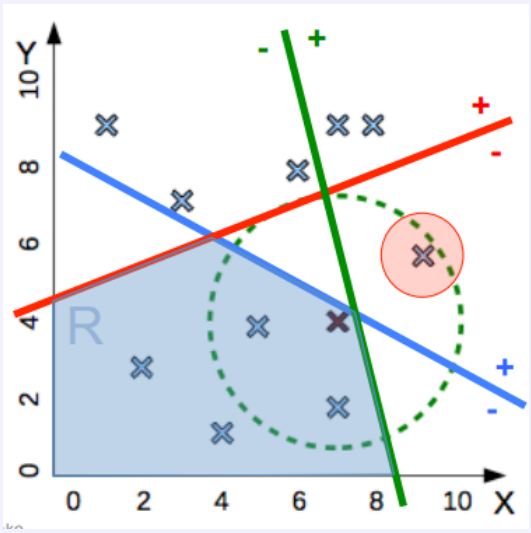
binary tree hierarchy where each two children corespond to a split on one of the axis (think decision trees)



time complexity of knn with this: $O(dlg(n))$, better for lower dimensional data, **can miss local neighbours**. Works best when n is much larger than 2^d

Locality-sensitive hashing

Divides space via k hyper planes, into 2^k regions (each divides space in two)

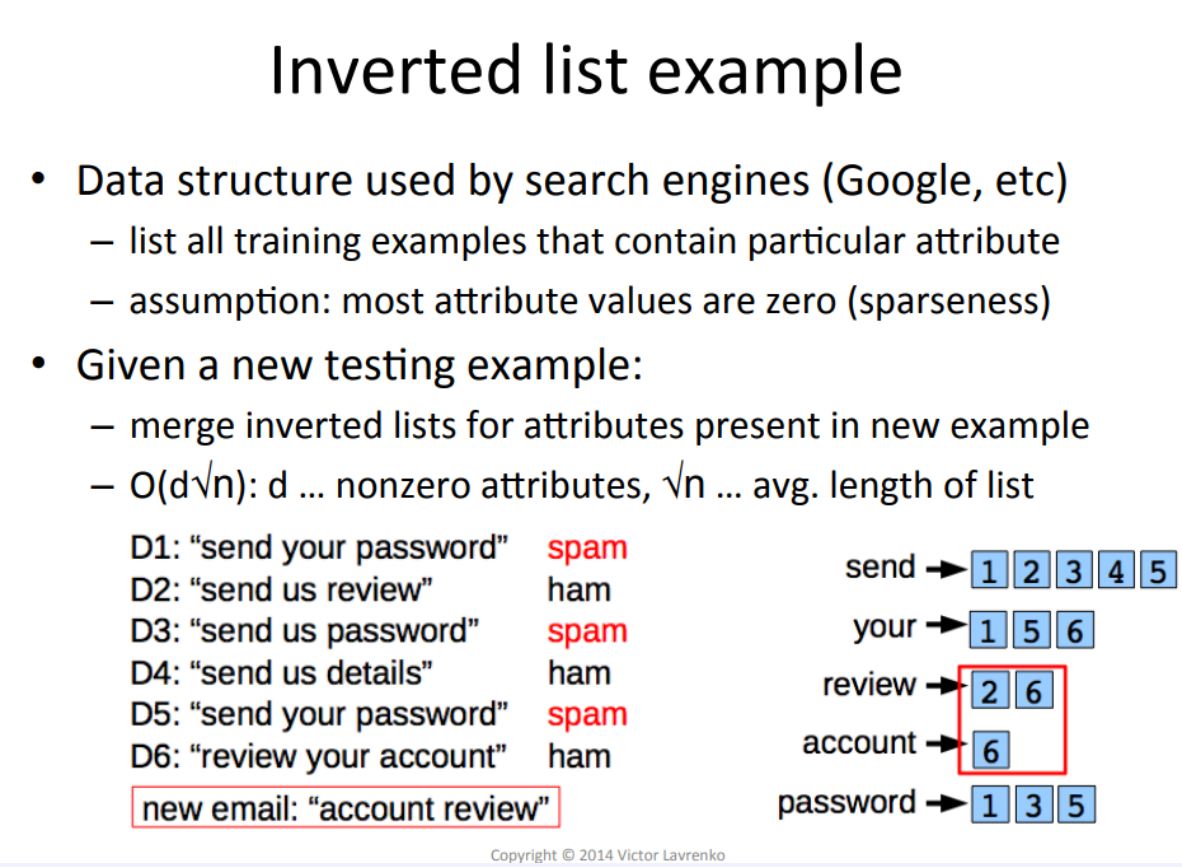


time complexity of knn with this: $O(kd + \frac{dn}{2^k}) \approx O(d \log n)$, **can miss local neighbours, but can remedy it by repeating with different hyperplanes**. Regions only tied to k hyperplanes not all the dimensions, better if data as a low-dimensional manifold structure (i.e. does not realy use all dimensions)

kNN evaluation

+ model-free, almost zero assumptions about the data apart from:
  - smoothness: nearby regions imply same class
  - assumptions implied by distance function locally
- need to handle missing data: perhaps fill in with mean or create a special distance
- sensitive to class-outliers (mislabeled training instances, perhaps by human error)
- sensitive to lots of irrelevant attributes (irrelevant attributes will affect the distance)
- very expensive computationally: space - all training samples, time - O(nd)
- expense is at testing not training (which doesn't exist)

Inverted list



Only really works for sparse data, otherwise list grows as big as the number of data points and is useless

K-Means

Clustering

Clustering is the process of discovering the underlying structures in the data. It is an **unsupervised** task

Types of clustering

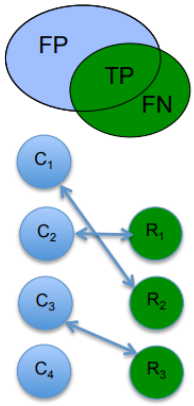
- Goal:
  - Monothetic: cluster based on some common property (i.e. all males aged 20-35)
  - Polythetic: cluster members are similar to each other (i.e. based on distance)
- Overlap:
  - hard clustering: clusters do not overlap
  - soft clustering: clusters may overlap
- Structure:
  - flat: clusters do not have any structure
  - hierarchical: clusters have parent clusters and form taxonomies

Evaluating clustering algorithms

- Extrinsic - helps us solve another problem. examples:
  - represent images with cluster features
  - train different classifier for each sub-population
  - identify and eliminate outliers
- Intrinsic - useful in and of itself
  - helps understand the makeup of our data
  - clusters correspond to classes
  - compare to human judgements

Intrinsic Evaluation

Intrinsic Evaluation 1



- System produces clusters C_1 C_2 … C_K
- Reference clusters (classes) R_1 R_2 … R_N
- Align up $R_i \Leftrightarrow C_j$, measure accuracy, F1, …
  - many different ways to align:
    - Weka: $C_j \rightarrow R_i$ with max overlap
  - if many $C_j \rightarrow$ same R_i :
    - re-assign in a greedy manner
    - non-greedy: K!/(N-K)! ways (very slow)
  - can we have multiple $C_j \rightarrow$ same R_i?
  - can we have multiple $R_i \rightarrow$ same C_j?
  - can we have overlapping clusters?

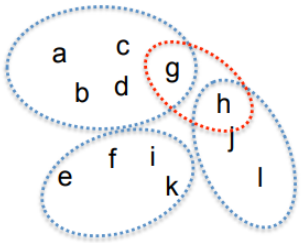
true class →	R2	R1	R3	
cluster → C1	3	1	2	6
C2	0	0	1	1
C3	7	1	8	16
C4	2	0	1	3
	12	2	12	

Accuracy = (3+0+8)/26

Intrinsic Evaluation 2

- Sample pairs x_i, x_j
  - ask human if x_i, x_j should be in the same group
  - easy task (cognitively)
  - can't ask them to "cluster" dataset manually

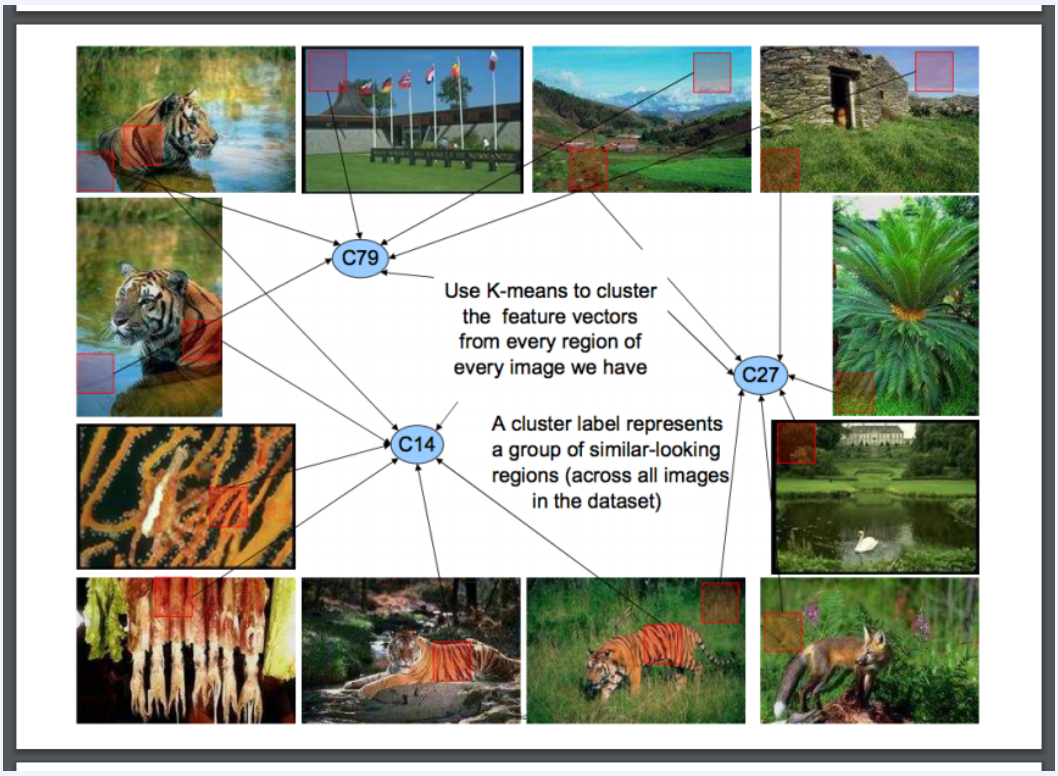
a,b = Yes
c,d = No
e,h = Yes
g,h = No



- System produces clusters
- Count errors, compute accuracy, F1, etc
  - FN: matching pairs x_i, x_j that are in different clusters (e,h)
  - FP: non-matching pairs x_i, x_j that are in same cluster (c,d)
- Doesn't require a pairing strategy
- Can handle overlapping clusters (a bit tricky)
  - same pair can count as both TN and FP (g,h = No)
- Can generate pairs from classes

Application in image classification

We can divide image into a set of patches, put all their features in a bag, and perform clustering on those bags, similar bags end up in similar clusters, name those something and use these as features for another algorithm!



K-means clustering

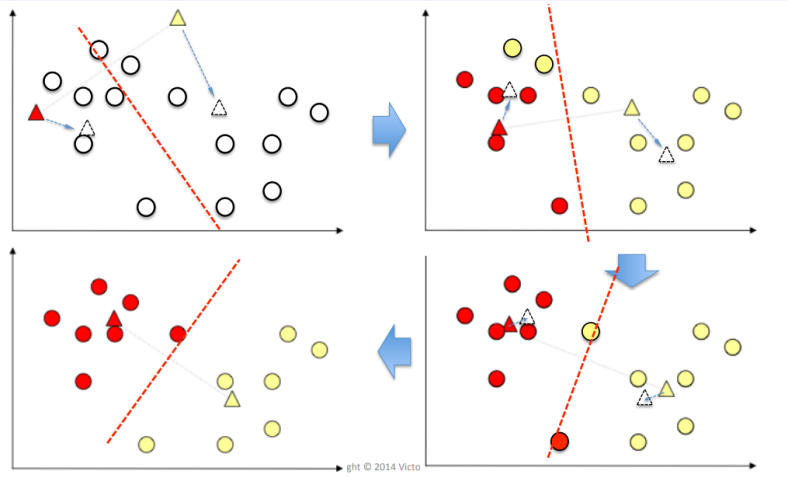
Minimizes intra-cluster distance:

$$\sum_j \sum_{x \to c_j} D(c_j, x)^2 \tag{65}$$

(same as variance if euclidian distance is used)
Always converges to a local minimum (meaning different starting points produce different results)

Algorithm

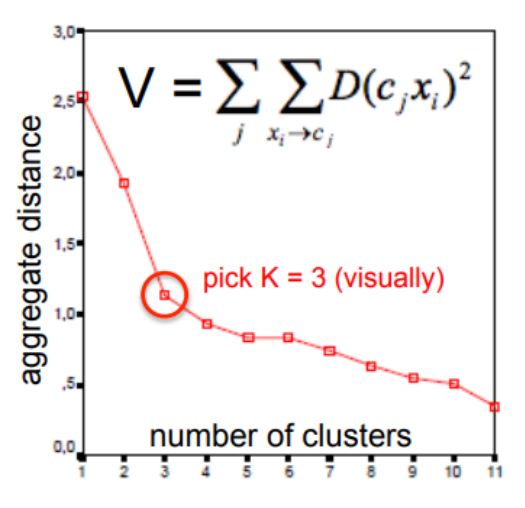
- place centroids at random locations
- untill convergence:
  - assign each point to nearest centroid
  - re-calculate each centroids position based on the mean of the positions of points assigned to it now



O(#iterations * #clusters * #instances * #dimensions)

Picking k

- No great algorithm for it, bad question to ask, you likely should know this.
- class labels may suggest it
- Optimize the total distance of points to their centers against k
- visually from scree plot (where the mountain ends and rubble begins)
- even if you use a validation set and try to pick best k, samples are still representative and so a clustering with higher k will likely end up with a lower distance in all cases



K-means evaluation

+ fast, iterative method
- converges to local minimum
- need to pick K
- need to pick distance function
- nearby points may end up in different clusters

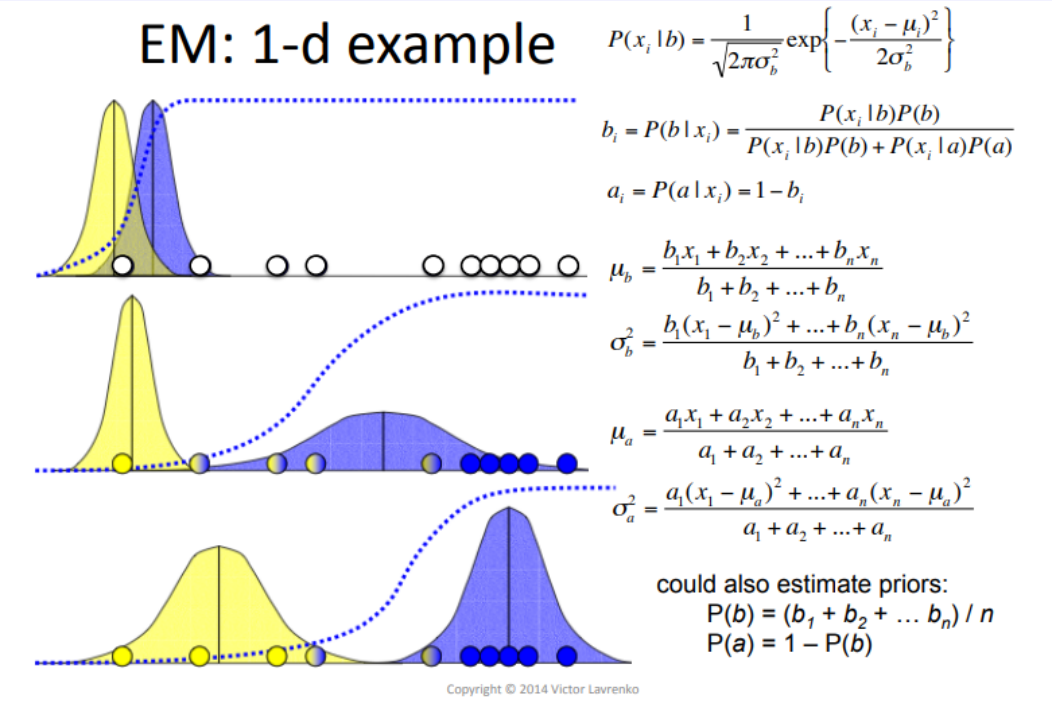
Gaussian mixture models

Mixture models

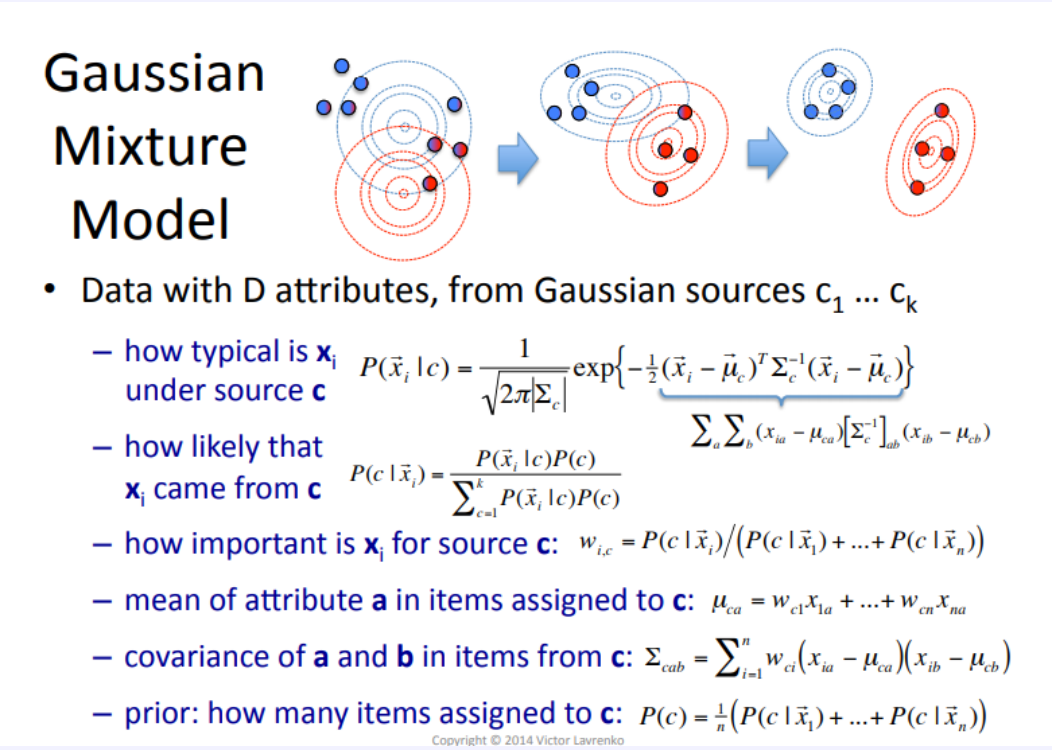
Maximizes likelihood of the data $P(x_1 \dots x_n)$

2 Gaussians

- start with two randomly placed gaussians (random parameters)
- calculate the probability P(b—x) that each data point comes from one of the gaussians (the other probability is 1-that)
- assign the points to the gaussians they're most likely from
- re-calculate the parameters of the gaussians from points assigned to them (MLE)
- repeat untill convergence



Multiple gaussians



the a-priori are the mixing coefficients, i.e. the proportions of each gaussian (these can be arbitrarilly set)

How to pick K

- Maximize the total likelihood based on k ($L = logP(x_1 \dots x_n) = \sum_{i=1}^{n} \sum_{k=1}^{k} P(x_i|k)P(k)$)
- K = n ? may not work well for new data points
- Pick best parameters on validation set (same issue as with k-means, k=n might still be best on a separate validation set if representative)
- Ocam's razor, pick simplest model that fits (Bayes inf. criterion)

Evaluation of mixturem models

K-means is a soft clustering algorithm - an instance can come from many different clusters, we can decide what to do in between gaussians.

- converges to local minimum
- need to pick k
  robust but slowe
+ k-means can only create circular boundaries while this can have arbitrary distributions (full covariance matrices)
- If you remove the co-variance result will be very similar to k-means

Principal components analysis

Hierarchical Clustering

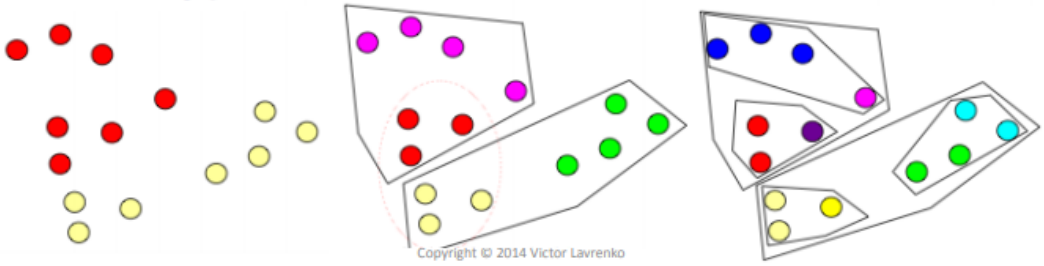
Hierarchical clustering

Picking k is a difficult task for clustering, why not simply find all possible clusters in a hierarchical structure ? This way you can evaluate the clusters at different granularities.

- top-down: start with all items in one cluster, split recursively
- bottom-up: start with singletons, merge with some criterion

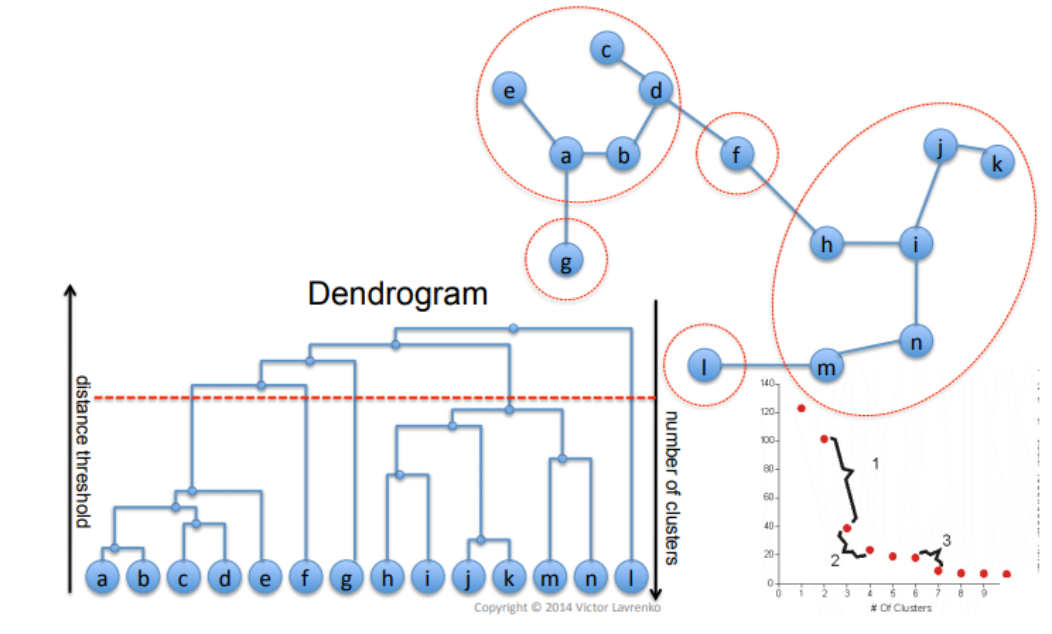
Hierarchical K-means

- Top-down approach:
  - run K-means algorithm on the original data $x_1 \ldots x_n$
  - for each of the resulting clusters c_i: i = 1 … K
    - recursively run K-means on points in c_i
- Fast: recursive calls operate on a slice: $O(Knd \log_K n)$
- Greedy: can't cross boundaries imposed by top levels
  - nearby points may end up in different clusters



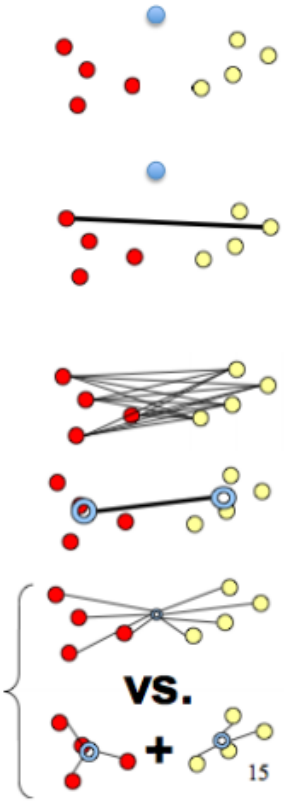
Agglomerative clustering

- Idea: ensure nearby points end up in the same cluster
- Start with a collection C of n singleton clusters
  - each cluster contains one data point: $c_i=\{x_i\}$
- Repeat until only one cluster is left:
  - find a pair of clusters that is closest: $\min_{i,j} D(c_i, c_j)$
  - merge the clusters c_i, c_j into a new cluster c_{i+j}
  - remove c_i,c_j from the collection C, add c_{i+j}
- Produces a dendrogram: hierarchical tree of clusters
- Need to define a distance metric over clusters
- Slow: $O(n^2d + n^3)$ – create, traverse distance matrix



Cluster distance measures

- Single link: $D(c_1,c_2) = \min_{x_1 \in c_1, x_2 \in c_2} D(x_1,x_2)$
  - distance between closest elements in clusters
  - produces long chains a→b→c→…→z
- Complete link: $D(c_1,c_2) = \max_{x_1 \in c_1, x_2 \in c_2} D(x_1,x_2)$
  - distance between farthest elements in clusters
  - forces "spherical" clusters with consistent "diameter"
- Average link: $D(c_1,c_2) = \frac{1}{|c_1|}\frac{1}{|c_2|}\sum_{x_1 \in c_1}\sum_{x_2 \in c_2} D(x_1,x_2)$
  - average of all pairwise distances
  - less affected by outliers
- Centroids: $D(c_1,c_2) = D\left(\left(\frac{1}{|c_1|}\sum_{x \in c_1}\vec{x}\right),\left(\frac{1}{|c_2|}\sum_{x \in c_2}\vec{x}\right)\right)$
  - distance between centroids (means) of two clusters
- Ward's method: $TD_{c_1 \cup c_2} = \sum_{x \in c_1 \cup c_2} D(x,\mu_{c_1 \cup c_2})^2$
  - consider joining two clusters, how does it change the total distance (TD) from centroids?



Lance-Williams Algorithm

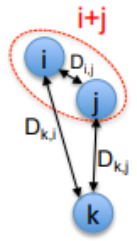
- D = {$D_{i,j}$: distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ for i,j=1..N}
- for N iterations:

  i,j = **arg min** $D_{i,j}$ … pair of closest clusters

  add cluster: i+j, delete clusters i, j

  for each remaining cluster k:

$$D_{k,i+j} = \alpha_i D_{k,i} + \alpha_j D_{k,j} + \beta D_{i,j} + \gamma |D_{k,i} - D_{k,j}|$$



Method	α_i	α_i	β	γ
Single linkage	0.5	0.5	0	-0.5
Complete linkage	0.5	0.5	0	0.5
Group average	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_j}$	0	0
Weighted group average	0.5	0.5	0	0
Centroid	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_j}$	$\frac{-n_i \cdot n_j}{(n_i+n_j)^2}$	0
Ward	$\frac{n_i+n_k}{(n_i+n_j+n_k)}$	$\frac{n_j+n_k}{(n_i+n_j+n_k)}$	$\frac{-n_k}{(n_i+n_j+n_k)}$	0

Single link:

$D_{k,i+j}$
= ½ (D_{ki} + D_{kj} - |D_{ki}-D_{kj}|)
= min {D_{ki} , D_{kj}}

$\min_{a,b} = \max_{a,b}$ - |a-b|

Perceptrons

Perceptron

A "neural" inspired model, single neuron with an activation function, predict as follows:

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} + w_0 \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{66}$$

Learning rule

Cannot differentiate the step function above so we use a training algorithm similar to gradient descent:

repeat
  for i in $1, 2, \ldots n$
    $\hat{y} \leftarrow \text{sign}[\mathbf{w}^T\mathbf{x}_i]$
    if $\hat{y} \neq y_i$
      $\mathbf{w} \leftarrow \mathbf{w} + y_i\mathbf{x}_i$
until all training examples correctly classified

Each time we move the boundary to face more towards the "real" label of the each data point.
If the data is linearly separable, the above algorithm always converges to a weight vector that separates the data.

Neural networks

Neural Networks

Neural networks are feature extractors. Multilayer networks are a **nonlinear** model.

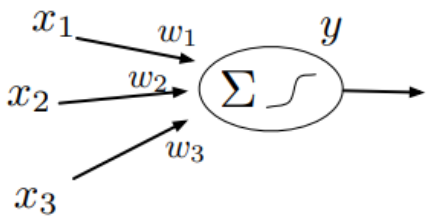
	Supervised		Unsupervised	
	Class.	Regr.	Clust.	D. R.
Naive Bayes	✓			
Decision Trees	✓			
k-nearest neighbour	✓			
Linear Regression		✓		
Logistic Regression	✓			
SVMs	✓			
k-means			✓	
Gaussian mixtures			✓	
PCA				✓
Evaluation				
ANNs	✔	✔		

Artificial Neural Networks

Inspired by the brain. **Feed forward** networks don't have cycles.

Single neuron

Each neuron takes the dot product of its weights with the input from the previous layer or directly from the input data, and as output produces the result of its activation function:



Take a single input $\mathbf{x} = (x_1, x_2, \ldots x_d)$. To compute a class label

1. Compute the neuron's activation
   $a = \mathbf{x}^\top \mathbf{w} + w_0 = \sum_{d=1}^{D} x_d w_d + w_0$
2. Set the neuron output y as a function of its activation
   $y = g(a)$. For now let's say

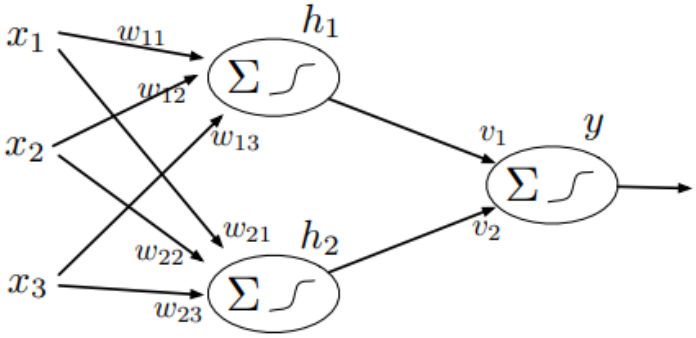
$$g(a) = \sigma(a) = \frac{1}{1 + e^{-a}} \qquad \text{i.e., sigmoid}$$

3. If $y > 0.5$, assign $\mathbf{x}$ to class 1. Otherwise, class 0.

Notice with the sigmoid activation function, this is just logistic regression.

Multilayer

When we connect the outputs of one layer to the inputs of another, we create multi-layer neural networks. These can get really complex



Here is how to compute a class label in this network:

1. $h_1 \leftarrow g(\mathbf{w}_1^T \mathbf{x} + w_{10}) = g(\sum_{d=1}^{D} w_{1d} x_d + w_{10})$
2. $h_2 \leftarrow g(\mathbf{w}_2^T \mathbf{x} + w_{20}) = g(\sum_{d=1}^{D} w_{2d} x_d + w_{20})$
3. $y \leftarrow g(\mathbf{v}^T \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} + v_0) = g(v_1 h_1 + v_2 h_2 + v_0)$
4. If $y > 0.5$, assign to class 1, i.e., $f(\mathbf{x}) = 1$. Otherwise $f(\mathbf{x}) = 0$.

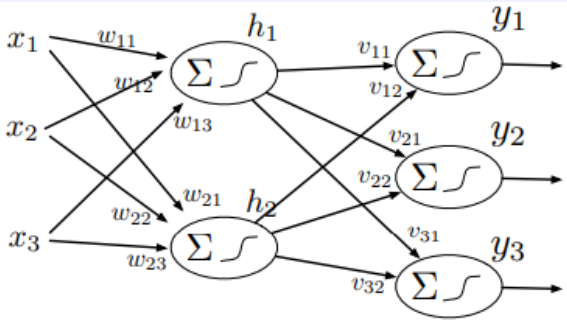
Each unit neuron gets its own weight vector (including bias) We can refer to all the weights stacked as one vector as $\mathbf{w}$

Regression

We can also perform regression with neural networks by simply taking the final output of the network directly instead of using an activation function

Multi-class classification

We can simply have more output neurons each corresponding to a class (could use a softmax layer to pick the class)



More than two classes? No problem. Only change is to output layer. Define one output unit for each class.

$y_i \leftarrow$ how likely it is that $\mathbf{x}$ in class i, $(i = 1 \ldots M)$

Then convert to probabilities using a softmax function.

$$p(y = m|\mathbf{x}) = \frac{e^{y_m}}{\sum_{k=1}^{M} e^{y_k}}$$

Parameters

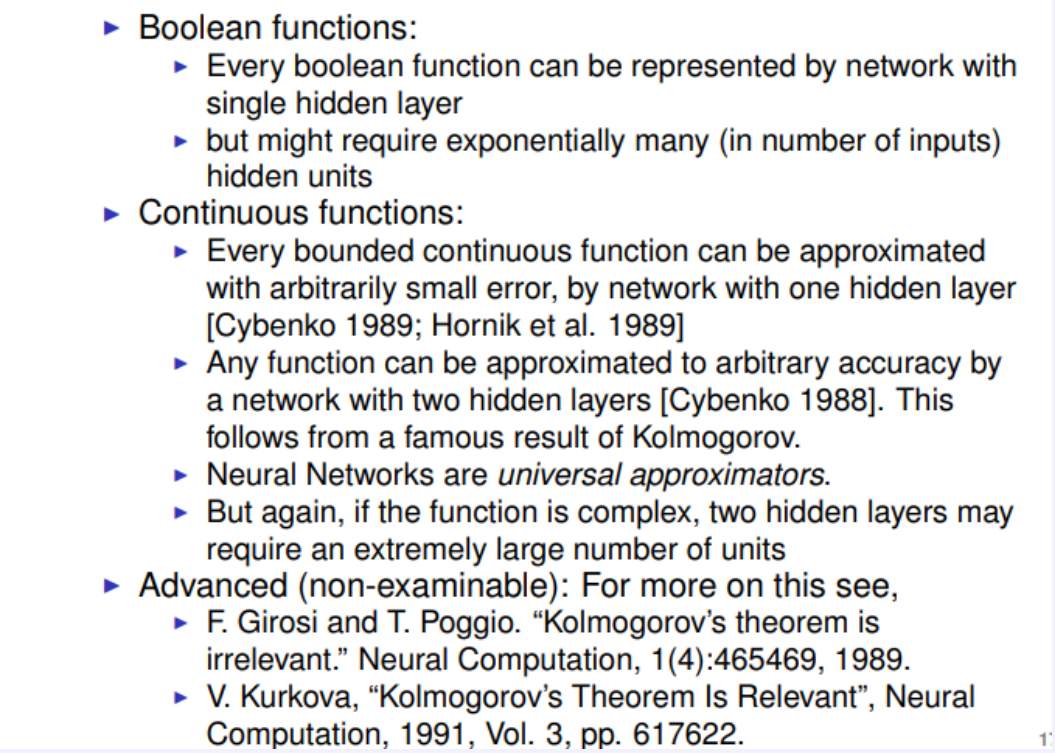
Neural networks are a black art, so many parameters, how many hidden layers, which activation functions, how many neurons in each layer ?

$$g(a) = \sigma(a) \quad \text{i.e., sigmoid}$$
$$g(a) = \tanh(a)$$
$$g(a) = a \quad \text{linear unit}$$
$$g(a) = \text{Gaussian density} \quad \text{radial basis network}$$
$$g(a) = \Theta(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases} \quad \text{threshold unit}$$

Note: using less hidden units than features, is analogous to dimensionality reduction

Representation power

Any function can be represented by a neural network with one to two hidden layers (Although for some they might need immensely many units). For boolean functions think CNF expressions.



Training

How do we find the weights for each unit? Create an error function compute gradient, akin to gradient descent but this time much more complex.
Can use **backpropagation**, a clever recursive algorithm which computes the derivatives of error using the chain rule (weighing the contribution of each unit to the error). With the gradient we can use any optimization routine to minimize E, we can adjust the weights as we please.
Note: this is absolutely non-convex, hitting the global optimum will reuqire starting the training with different starting weights

Evaluation of NNs

+ non-linear, can model really complex functions
+ can be used to extract features to be used elsewhere (each hiden layer contains some sort of highly non-linear feature based on the input)
- converge to non-optimal weights