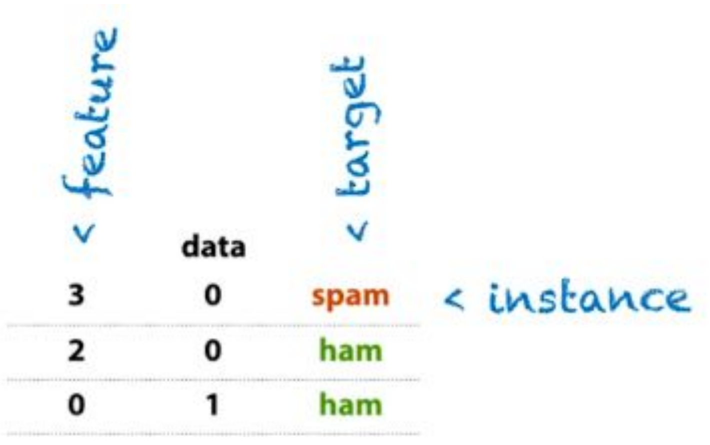
Lecture 1: Introduction

Machine learning provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Usually deals with offline learning > train model once and then it's done. Then use this model.

When to use ML?
• we can’t solve it explicitly. • approximate solutions are fine • plenty of examples available. (for example: recommendation systems for movies)

ML allows us to learn programs that we have no idea how to write ourselves. Machine learning allows us to create programs from a set of examples.

<u>Supervised learning (labeled data/have examples)</u>
1. Classification
instances = data example line
features (of the instances) = things we measure (numeric/categorical)
target (value) = what we are trying to learn


< feature
< target
< instance

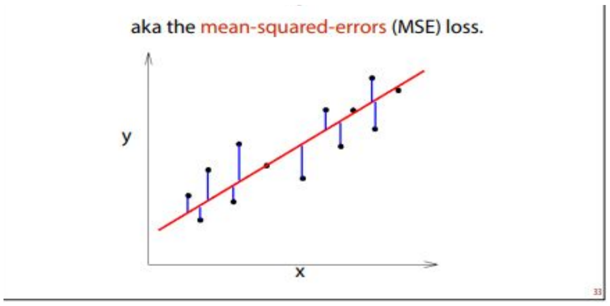
	data	
3	0	spam
2	0	ham
0	1	ham

Example 1: Linear classifier
loss(model) = performance of model on the data (the lower the better) for classification: e.g. the number of misclassified examples. used to search the model space. Input: model, has data as constant.
Example 2: A decision tree classifier = studies one feature in isolation at every node.
Example 3: K-nearest neighbours: lazy: for a new point, it looks at k points that are closest (k=7 f.e.) and assigns the class that is most frequent in that set. k is what we call a **hyperparameter**: you have to choose it yourself before you use the algorithm. Trial & error/grid search/random search

Variations:
• Features: usually numerical or categorical.
• Binary classification: two classes, usually negative and positive (pos = what you are trying to detect)

• Multiclass classifcation: more than two classes
• Multilabel classifcation: more than two classes and none, some or all of them may be true
• Class probabilities/scores: the classifer reports a probability for each class.

2. Regression
Loss function for regression: the Mean-squared-errors (MSE) loss → Measure the distance to the line, this is the difference between what the model predicts and the actual values of the data. Take all values and square them: so they are all positive (& so they don't cancel each other out). Sum them up, and then divide by size of dataset (average). the lower MSE, the better (blue line residual)
Assumes normality, so sensitive to outliers

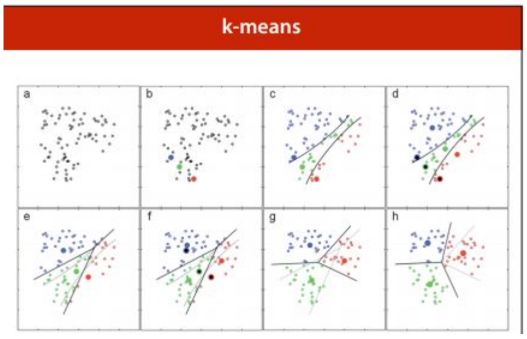


Example 1. Linear regression (straight line)
Example 2. Regression tree (go through every point)
Example 3. kNN regression (take K=x closest points)

Grouping models segment the feature space. Grading models can assign each element in the feature space a different prediction. Grouping models can only assign a finite number of predictions.
Grouping model ROC curves have as many line segments as there are instance space segments in the model; grading models have one line segment for each example in the data set. This is a concrete manifestation of something I mentioned in the Prologue: grading models have a much higher 'resolution' than grouping models; this is also called the model's refinement. by decreasing a model's refinement we sometimes achieve better ranking performance.

Overfitting = Our model doesn't generalize well from our training data to unseen data; it draws too any specific conclusions from the training data. If our model does much better on the training set than on the test set, then we're likely overfitting.
~Split your test and training data!~
Aim of ML is to not to minimize loss on training data, but to minimize on test data.
How to prevent? Never judge our model on how well it does on the training data.We withhold some data, and test the performance on that. The proportion of test dat you withhold is not very important. It should be at least 100 instances, although more is better. To avoid overfitting, the number of parameters estimated from the data must be considerably less than the number of data points.

<u>Unsupervised learning tasks</u>(unlabeled data)
1. Clustering → split the instances into a number of (given) clusters. Example of clustering algorithm: **K-means**. In the example we will separate the dataset shown in (a) into three clusters. It starts by picking 3 main points, and color them by the mean color they are close to. Do this again, and throw away old coloring. Keep doing this until done.



2. **Density estimation** → when we want to learn how likely new data/examples is. Is a 2 m tall 16 year old more or less likely than a 1.5 m tall 80 year old? (normal distribution simple form of density estimation)

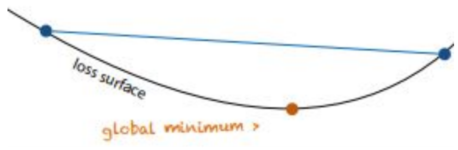
3. Generative modelling (sampling)
With complex models, it's often easier to sample from a probability distribution that it is to get a density estimate. Sample pictures to get new sample.

Lecture 2: Linear models 1

Optimization= trying to find the input for which a particular function is at its optimum (in this case its minimum value)

Random search = pick a random point and pick a point quite close to it and see which one is better. If the new point is better, move to this new point and go again, if new point isn't better, you discard it. Sensitive to local minimum

Convex= if you pick any two random points on the loss surface and draw a line between them, everything in between those points need to be below that line: practically means that we have 1 (global) minimum and this minimum is the optimal model. So long as we know we're moving down (to a point with lower loss), we can be sure we're moving in the direction of the minimum.

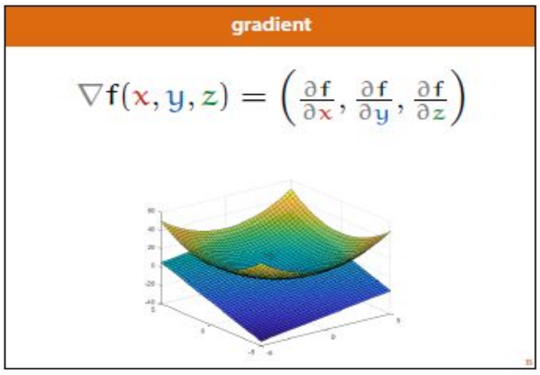


What if the loss surface has multiple local minima?
1. Simulated annealing = similar to random search but little difference: if the next point chosen isn't better than the current one, we still pick it, but only with some small probability P. In other words, we allow the algorithm to occasionally travel uphill. This means that whenever it gets stuck in a local minimum, it still has some probability of escaping, and finding the global minimum.

→ Random search & simulated annealing: black box optimization (--> don't need to know specific information/insight/compute gradient about model, only need to compute/evaluate loss function)
Features: very simple • can require many iterations (takes long, can get stuck in local minimum) • also works for **discrete** model spaces

2. Run random search a couple of times independently. One of these runs may start you off close enough to the global minimum. For simulated annealing, doing multiple runs makes less sense since it doesn't get stuck. If you wait long enough, it will find it.
To escape local minima→ add randomness (SA)
To converge (= find certain point) faster → inspect the local neighbourhood (to determine in which direction the function decreases quickest)

Gradient descent: start with a random point, we compute the gradient and subtract it from the current choice (because the gradient is the direction of steepest descent that we want to go downhill) and iterate this process. only for **continuous** models



gradient

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$$

Since the gradient is only a linear approximation to our loss function, the bigger our step the bigger the approximation error. Usually we scale down the step size indicated by the gradient by multiplying it by a learning rate η. This value is chosen by trial and error, and remains constant throughout the search. If our function is non-convex, gradient descent doesn't help us with local minima → add a bit of randomness

Sometimes your loss function should not be the same as your evaluation function.

Loss functions serve two purposes:

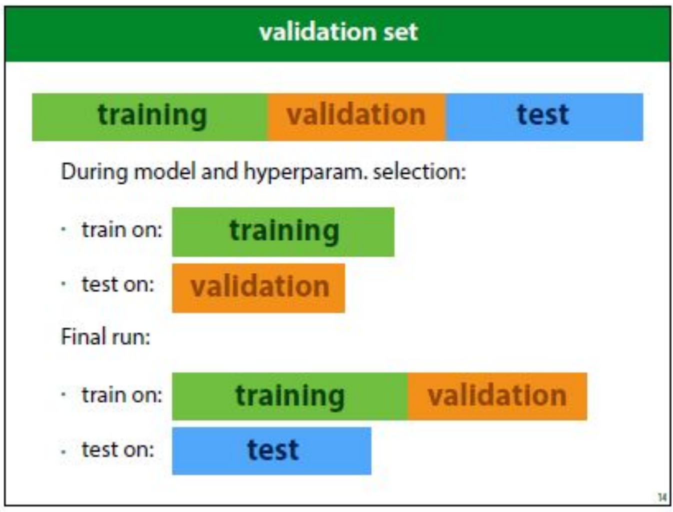
1. to express what quality we want to maximise in our search for a good model
2. to provide a smooth loss surface(so that the search for a minimum can be performed efficiently)

Lecture 3: Methodology 1

Class imbalance= the proportion of the positive class is so small in relation to the negative class that the accuracy doesn't mean anything. For example: you create a classification model and get 90% accuracy immediately, but you discover that 90% of the data belongs to one class. Do not assume a high accuracy is a good accuracy!

Cost imbalance= the cost of getting it wrong way one way vs the other is very different. (diagnosing a healthy person with cancer (lower) vs. diagnosing a person with cancer as healthy (higher)) Both come with a cost but not the same cost (spam vs. ham)

The simplest and most useful sanity check for any machine learning research, is to use **baselines** → a simple approach to your problem to which you compare your results: it helps to calibrate your expectations for a particular performance measure on a particular task.



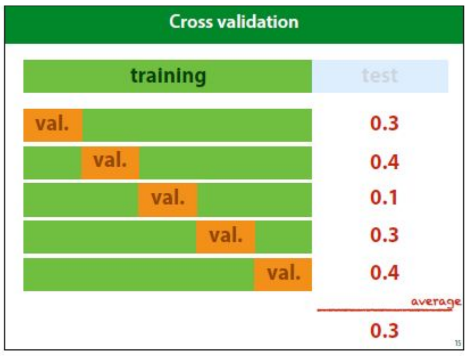
validation set

Hyperparameters are the parameters that are chosen, not learned from the data.

How do we choose the hyper parameter? Ideally, we try a few and pick the best. However, it would be a mistake to use the test set for this.

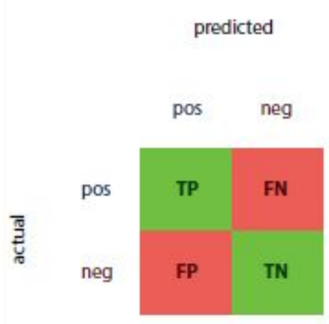
Different tests for accuracy may give different results, because of too small test data or testing too many different things on one test set.

After you've split off a test and validation set, you may be left with very little training data → **Cross validation**= You split your data into 5 chunks and for each specific choice of *hyperparameters* that you want to test, you do five runs: each with one of the folds as validation data. You then average the scores of these runs → This can be costly (for every parameter test and train 5 times), but you ensure that every instance has been used as a training example once.

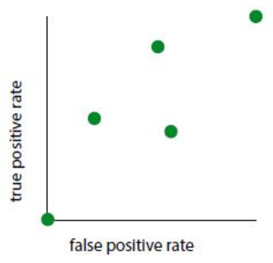


Confusion matrix/ contingency table

We call accurately classified instances true positives and true negatives. Misclassifications are called false positives and false negatives.
true positive rate: what proportion of the actual positives did we get right
false positive rate: what proportion of the actual negatives did we get wrong (by labelling them as positives)



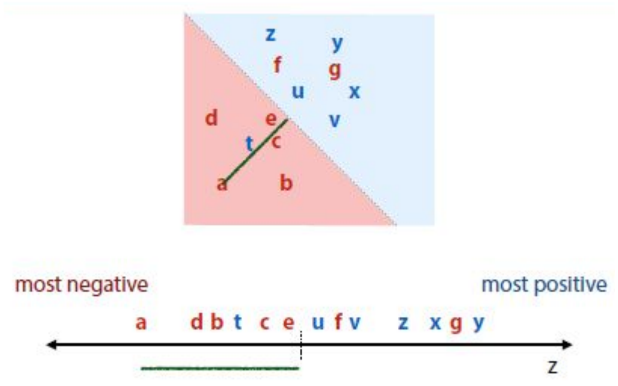
ROC space

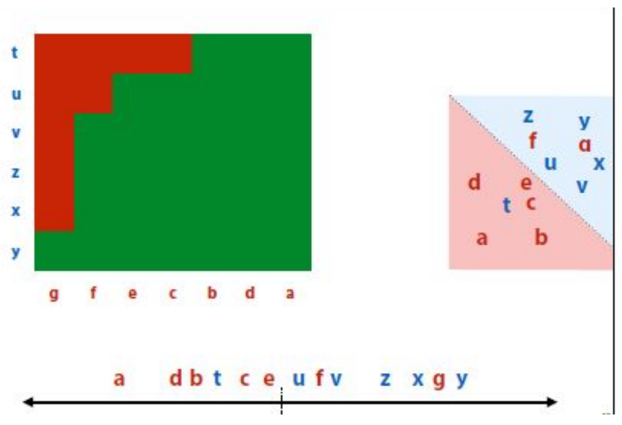


Bottom left corner: classifies nothing as true.
Top right corner: classifies everything as true. Every dot is a classifier

Ranking/scoring classifier → doesn't just provide classes, it also give a score of how negative or how positive a point is. We can use this to rank the points from most negative to most positive, by measuring the distance to the decision boundary.

For every pair of one positive and one negative instance, the ranking classifier should rank the negative lower than the positive. If it fails to do so (like with the pair (z, g)) we call that a *ranking error*. We can now scale our classifier from timid to bold by moving the decision boundary from left to right

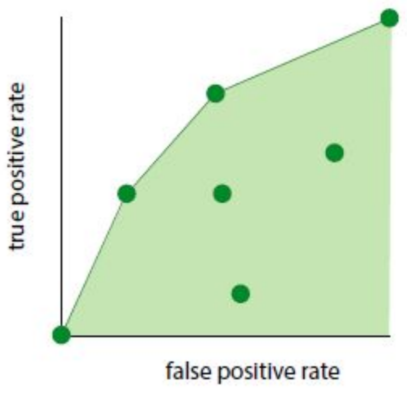




Coverage matrix = We can make a big matrix of all the pairs for which we know how they should be ranked: negative points on the horizontal axis, positive on the vertical. The more sure we are that a point is positive, the closer we put it to the bottom left corner. Matrix red = the probability of making a ranking error

Normalizing the coverage matrix gives us the ROC space. We can see that the area under the ROC curve traced out by the classifier is <u>an estimate of the (probability that we are going to make a ranking error)</u>
The bigger the AUC, the better the classifier.

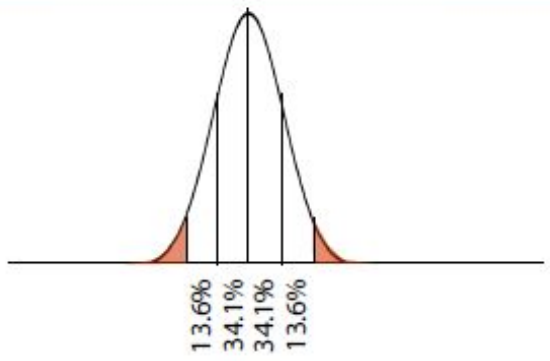
Accuracy is a metric for a single classifier. AUC is a metric for a *collection of classifiers*, usually from a ranking classifier → how to turn a classifier into a ranking classifier, depends on the type of classifier.
AUC is a good metric if we don't know the relative importance of the classes, or if the classes are unbalanced. (class/cost imbalance)
Finally, set a threshold for the ROC curve.



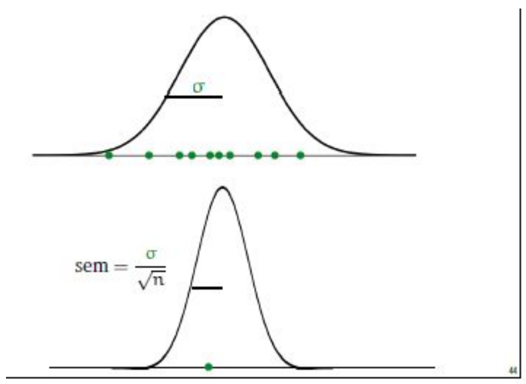
should we do statistical tests in ML at all?
• Makes ML experimentation difficult. Lots of disagreement.
• People overestimate the value of statistical analyses.
• Does not promote the best methods
• The ultimate validation of research is REPLICATION

Error bars most common options: - standard deviation - standard error - confidence interval
standard deviation = measure of *spread*, variance (sample more → more accurate
standard error, confidence interval = measures of confidence (more data → smaller)

Standard error (of the) mean (SEM)
If we sample data multiple times and calculate the mean (also a random variable, because if we sample again, we get a different mean). SEM is the distribution of the whole sampling process. The new distribution is related but more narrow since there is less variance.
SEM= standard deviation original distribution / square root n (sample size)
If original distribution is normal, so is the sample. The bottom distribution is a Student's-t distribution. For large enough sample sizes (more than 100), we can treat this as a normal distribution. One of the reasons that the normal distribution is so popular is that it has a definite scale.

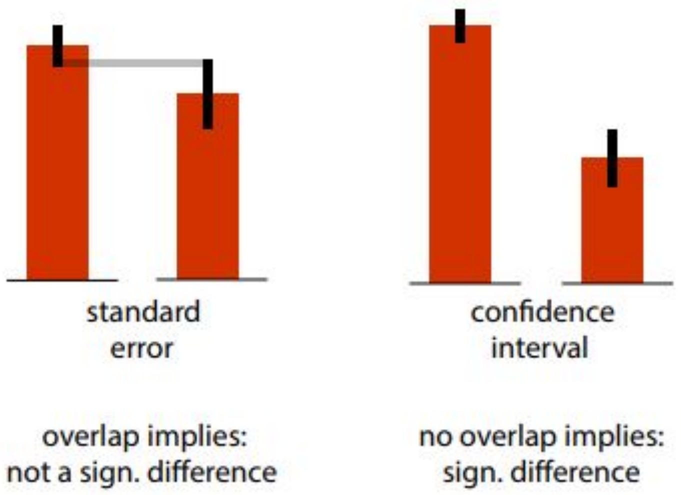


mean +/- 2 sem is a 95% confidence interval



Confidence intervals:
Don't say: the probability that the true mean is in the confidence interval is 95%.
Do say: If we repeat the experiment many times, computing the confidence interval from a random sample; the true mean would be inside the interval in 95% of those experiments.

If the SEM error bars do not overlap, there may or may not be a significant difference. If the confidence interval error bars do overlap, there may still be a significant difference, depending on how much they overlap



why use statistics in ML? • to show confidence • to show spread

bootstrapping = *sample with replacement* until you have a dataset of the same size as the whole dataset. Some points may be in there multiple times, once or not at all.
On average, about 63.2% of the dataset will be included. The rest will be duplicated instances. Each bootstrapped sample lets you repeat your experiment.
Note that some classifiers will respond poorly to the presence of duplicate instances! Better than cross validation for small(er) datasets.

The no-free-lunch theorem(s)
"… any two optimization algorithms(/classifier) are equivalent when their performance is averaged across all possible problems" → no single best classifier!
According to the NFL theorem, there are as many datasets X for which C beats D as there are for which D beats C.

The universal distribution
Not all datasets are created equal. The datasets for which our method works, are the likely ones. The universe "generates" data for which our methods work:
• Compressible data & Simple data
The datasets that don't work aren't selected, because they look random to us

No free lunch principle: '*There is no single best learning method. Whether an algorithm is good, depends on the domain'*

Occam's razor → "The simplest explanation is often the best" We should bias our algorithms towards simple models. It reduces overfitting, helps generalization

Lecture 4: Methodology 2

Cleaning your data: missing data & outliers
Solution:

- remove the feature(s) (f.e. whole column) for which there are values missing. If lucky, the feature is not important anyway.
- remove the instances (i.e. the rows) with missing data. If the data was not corrupted **uniformly,** removing rows with missing values will change your data distribution → for example, if only a certain group did not answer a question

When removing instances/features, think about the REAL-WORLD use case. Can you expect missing data there too, or will that data be clean already?

<u>Will you get missing values in production?</u>
YES: Keep them in the test set, and make a model that can consume them.
NO: Get a test set *without missing values*, and test different methods for completing the data.

<u>Guess the missing values (=imputation)</u>

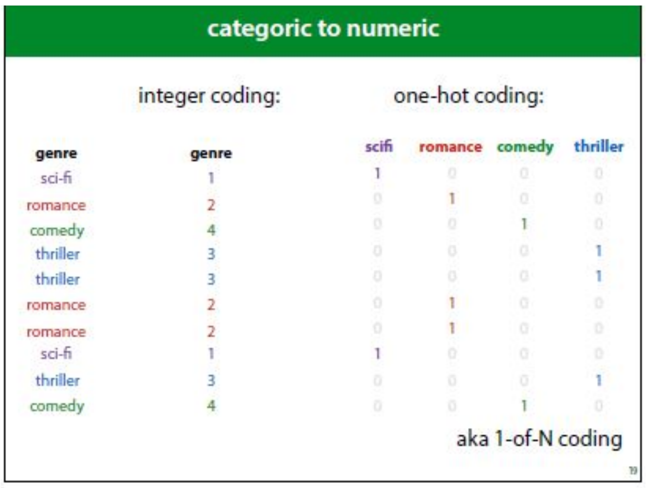
- ➔ categorical data: use the mode
- ➔ numerical data: use the mean
- ➔ make the feature a target value and train a model (kNN, linear regression, etc.)

Outliers can be measurement errors, or just extreme values. In the latter case, the model needs to be adapted.

Are they mistakes?
Yes: deal with them
No: leave them be. Check if your model assumes normality.
Can we expect them in production?
Yes: Make sure the model can deal with them (remove or work them into the model)
No: Remove. Get a test set that represents the production situation.

Even if your data comes in a (complete) table, that doesn't necessary mean that every column can be used as a feature right away (or that this would be a good approach).

So what if our model only accepts numerical features? How do we feed it categorical data?
1. Integer coding → imposing a false ordering on unordered data.
2. One-hot coding/1-of-N coding → turning one categorical feature into several numeric features.



categoric to numeric

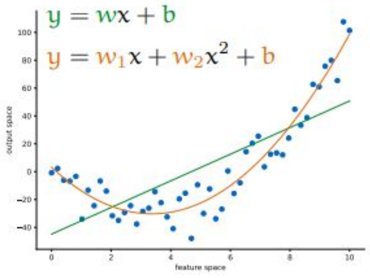
integer coding: one-hot coding:

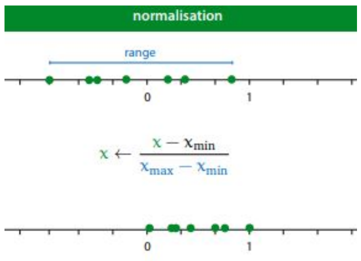
genre	genre	scifi	romance	comedy	thriller
sci-fi	1	1	0	0	0
romance	2	0	1	0	0
comedy	4	0	0	1	0
thriller	3	0	0	0	1
thriller	3	0	0	0	1
romance	2	0	1	0	0
romance	2	0	1	0	0
sci-fi	1	1	0	0	0
thriller	3	0	0	0	1
comedy	4	0	0	1	0

aka 1-of-N coding

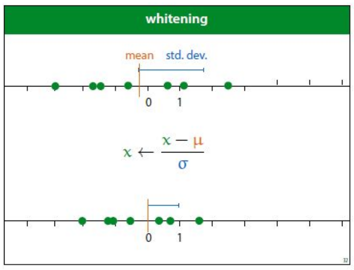
19

Per genre we can say whether it applies to the instance or not.
Linear classifiers/models are extremely simple to fit. It is a powerful model because it can fit many nonlinear aspects of the data. By adding features we can make it useable. If we don't have any intuition for which extra features might be worth adding, we can just add all cross products. Other functions may include the sine or the logarithm. (green line original, orange with added features)



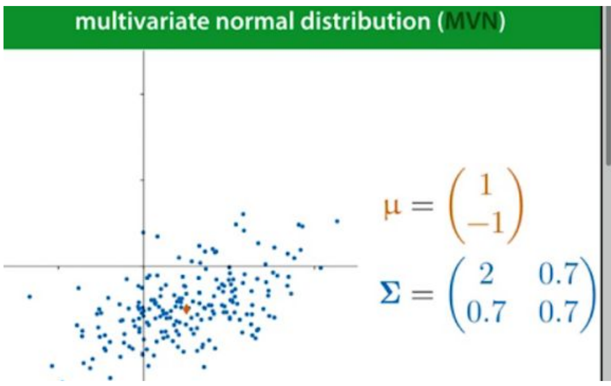


Normalization → scales the data linearly so that the smallest point becomes 0 and the largest becomes 1. Indep. for each feature
whitening → recall the data so that the mean becomes 0, and the standard deviation becomes 1 → transforming so it looks like a normal standard distribution.
If data uncorrelated → reduced to a nice spherical distribution, centered on the origin, with the same variance in each direction.



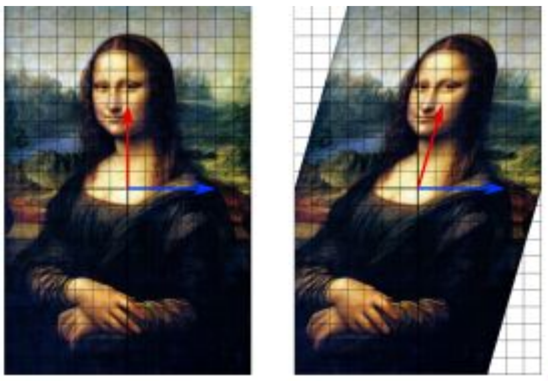
If data correlated --> we get a different result. This is because we whiten each feature independently, and the features are not independent.

multivariate normal distribution (MVN)= a generalisation of a one-dimensional normal distribution. Its mean is a vector (a single point) and the standard deviation/variance is determined by a symmetric matrix called a covariance matrix. The values on the diagonal indicate how much variance there is along each direction. The off-diagonal elements (left below and right up) indicate how much covariance/correlation there is between axis x and y (how diagonally stretched is the data).



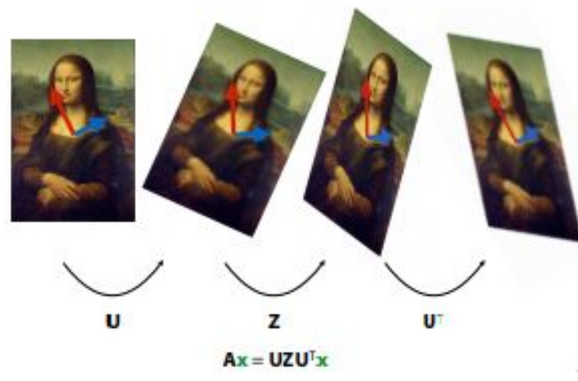
One very helpful way to think of MVNs is as transformations of a single standard MVN.
Why not use S=AA^T method? When rebasing a dataset, how the old axes map to the new axes is arbitrary (which of the new axes will be axis 1 is an arbitrary choice). We can achieve some very interesting effects if we make a more considered choice. Specifically, in the new coordinates, we'd like the axes to be ordered by variance: the first axis should be the one along which the variance is the highest.

Eigenvector= A vector is an eigenvector of a matrix if it only gets stretched (or flipped), but <u>its direction doesn't change (blue line is eigenvector).</u> A vector u is an eigenvector of A if multiplying A by u is the same as multiply u by some scalar lambda. Au = λu. For some matrices, the eigenvectors align with the axes. That is, the normal basis vectors are eigenvectors. Such matrices called **scaling**

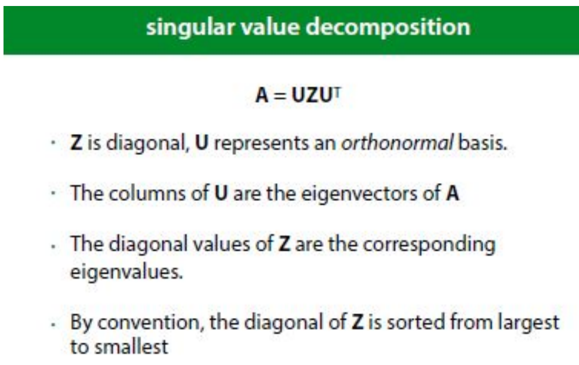


matrices are zero everywhere, except on the diagonal.$\begin{pmatrix} ⅓ & 0 \\ 0 & 4/3 \end{pmatrix}$

What if our transformation is not a scaling matrix?
1. Transform the picture so that the eigenvectors are aligned with the axis (rebase it) (u)
2. Apply a scaling matrix (z)
3. undo the change of basis (we require that U is orthonormal, so we can use UT) to return to the original coordinates → transform the picture back (to original base) (u^t)



We can decompose the matrix A into (product of three matrices) UZU^T
This is called **singular value decomposition** → decompose our matrix into these two matrices U&Z. We can search for U and Z through stochastic gradient descent. The last bullet point is important. This will ensure that in our new coordinates, the axis **with the greatest variance/stretch will be the first.**

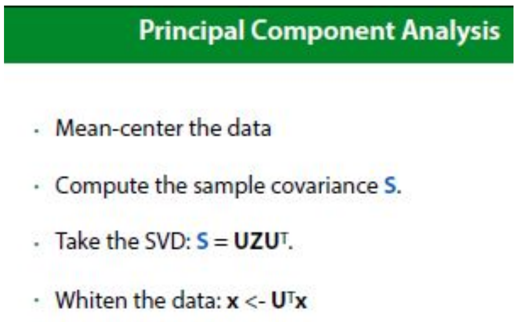


singular value decomposition

$A = UZU^T$

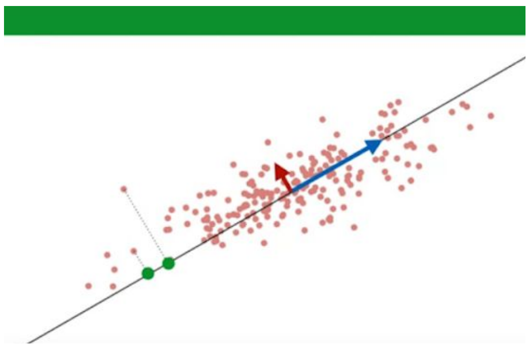
- **Z** is diagonal, **U** represents an *orthonormal* basis.
- The columns of **U** are the eigenvectors of **A**
- The diagonal values of **Z** are the corresponding eigenvalues.
- By convention, the diagonal of **Z** is sorted from largest to smallest

→ **Principal component analysis (PCA)** = a data normalisation method that takes feature correlations into account.
The first axis is the one with the highest eigenvalue = the direction in which our data has the most variance. Second axis/eigenvector has the second biggest eigenvalue, direction in which is the most residual variance. It is good for whitening the data, but also for reducing the number of dimensions.
If we represent the points of our data only by their first principal component, we project them onto this line. In some sense, this give us the best linear projection of our data onto one dimension.



Principal Component Analysis

- Mean-center the data
- Compute the sample covariance **S**.
- Take the SVD: $\mathbf{S} = \mathbf{UZU}^T$.
- Whiten the data: $\mathbf{x} \leftarrow \mathbf{U}^T\mathbf{x}$



• Expresses the data in new coordinates, aligned with the covariance.
• The first coordinate (first principal component) is the line along which the data has the most variance.
• The second coordinate is the line along which the remaining variance is the highest (so on)
• Representing data by only its first k principal components, is a great data reduction/compression method.

Lecture 5: Probabilistic models 1

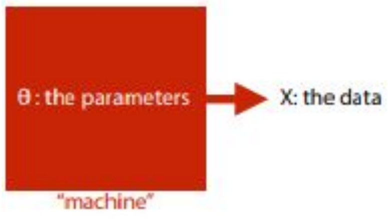
Two kinds of probability:
1. **In objective probability,** the probability that “X is the case” can only be defined as the outcome of repeated experiments. For instance, the probability of rolling 6 with a fair die, is one in six. Probability of X is the same for all people regardless of what we know or don’t know. (**frequentism**: probability is only a property of repeated experiments)

2. **Subjective probability** states that probability expresses our uncertainty. If X is a boolean variable, one that is true or not true, and we are uncertain whether X is true, we can assign a probability to X being true (x=0-1) (**Bayesianism**: probability is an expression of our uncertainty and of our beliefs.) Subjective view on the world and people

Sample space= These are the outcomes or truths that we wish to model (head/tails)
1. Discrete sample space = with limited values, 1-6 etc.
2. Continuous sample space= for example with R (real numbers)

Event space= all events that have probability
powerset= the set of all subsets

the inversion problem/bayes’ rule: It’s easy to express the probability of an observable given some hidden cause (assuming we have a model of the world). However, we usually want the opposite



Theta= parameter(s)
maximum likelihood principle/estimation = criterion used by frequentists. They say that the theta you should choose assuming the data, is the one that maximizes the formula (--> probability of the data given the parameter). You choose the configuration of the machine under which the data is most likely/for which the highest probability exists of seeing the data. Parameters don’t have a probability.

Bayesians do not care for point estimates, they express their uncertainty about the value of the parameter as a probability distribution. p(θ)|X)=posterior, p(θ)=prior

$$p(\theta \mid X) = \frac{p(X \mid \theta)p(\theta)}{p(X)}$$

The prior expresses beliefs in the values of the parameters before seeing the data (cannot be computed, based on actual beliefs), the posterior tells you your beliefs afterwards.

Maximum a Posteriori (MAP) model= halfway between frequentist and bayesian, method disowned by both. It is like the maximum likelihood model but with a prior. Take a point estimate but do take in account the prior probability of certain models/parameters. Often better than maximum likelihood principle. Multiply likelihood with prior.

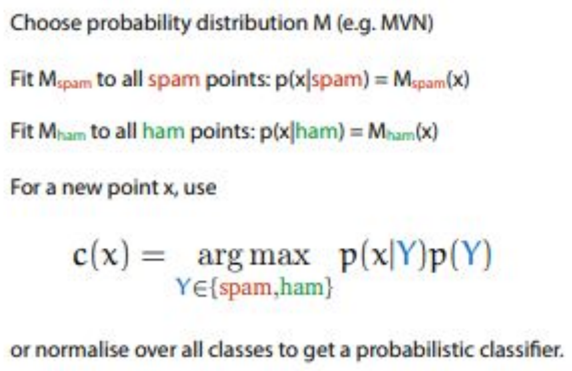
$$\hat{\theta} = \arg\max_{\theta} p(X \mid \theta)p(\theta)$$

probabilistic classifiers = classifiers that return not just a class for a given x (or a ranking) but a probability over all classes. Note that a probabilistic classifier is also immediately a

ranking classifier and a regular classifier. f.e. P(Y=spam|X) = 0.1, P(Y=ham|X) = 0.9.
We can use the probabilities to extract a ranking (and plot an ROC curve) or we can use the probabilities to assess how certain the classifier is. If we don't want the probabilities, we can just turn it into a regular classifier by picking the class with the highest probability.

two approaches:
1. Discriminative approach= *learn a function/class distribution for p(Y|X) directly*. It functions as a kind of regression, where we map x to a (feature) vector of class probabilities → discriminate between X's of one class and X's of other class.
2. Generative approach= *learn p(x|y) and p(y)*. p(Y|X) ∝ p(X|Y)p(Y). Also produces class probabilities, but goes the long way round. *It learns a distribution p(Y)* (which is usually easy to estimate from the data) and, more importantly, *it learns a distribution of points given the class* → generative because if you have this probability distribution then you can basically sample from it so you can sample class probabilities from this probability distribution.

Generative: Bayesian classifier:



Choose probability distribution M (e.g. MVN)

Fit M_{spam} to all spam points: $p(x|spam) = M_{spam}(x)$

Fit M_{ham} to all ham points: $p(x|ham) = M_{ham}(x)$

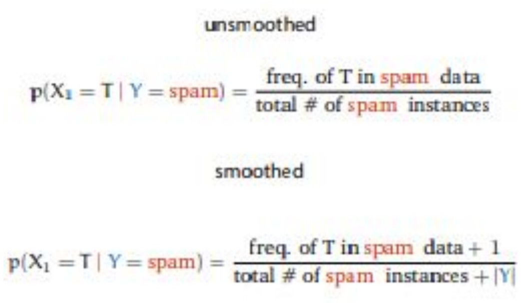
For a new point x, use

$$c(x) = \underset{Y \in \{spam, ham\}}{\arg\max}\ p(x|Y)p(Y)$$

or normalise over all classes to get a probabilistic classifier.

This works well for small numbers of features, but if we have many features, modelling the dependence between each pair of features gets very expensive. A crude, but very effective solution is **Naive Bayes=** just assumes that all features are independent, conditional on the class. p(X1, X2 | Y) = p(X1 | Y)p(X2 | Y) (generative)
It selects all emails of one class, and then estimates the probability that X1 will be T as the relative frequency of emails for which X1 was T in the training set. Do the same for all features. p(x=T|spam)=⅗, p(x=F|spam)=⅖
If thing in Naive Bayes 0, whole formula 0. To solve this:

Laplace smoothing: *add pseudo-observation*s to avoid zero probabilities. For each possible value, we add an instance where all the features have that value. for example, we add for spam & ham a row where both are T and both are F.

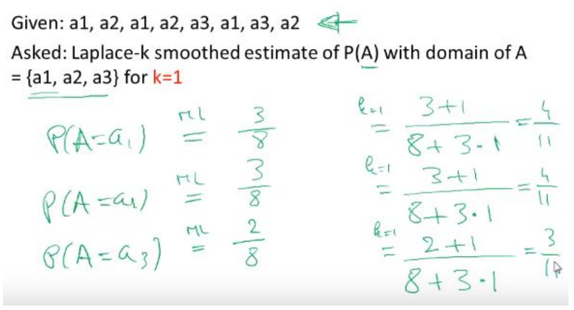


unsmoothed

$$p(X_1 = T \mid Y = spam) = \frac{\text{freq. of T in spam data}}{\text{total \# of spam instances}}$$

smoothed

$$p(X_1 = T \mid Y = spam) = \frac{\text{freq. of T in spam data} + 1}{\text{total \# of spam instances} + |Y|}$$



Given: a1, a2, a1, a2, a3, a1, a3, a2

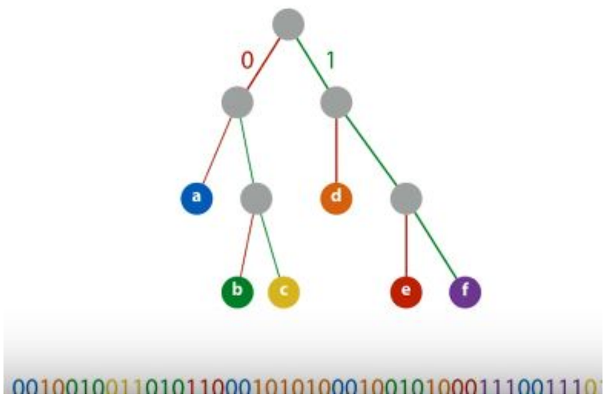
Asked: Laplace-k smoothed estimate of P(A) with domain of A = {a1, a2, a3} for k=1

$P(A=a_1) \overset{ML}{=} \frac{3}{8} \overset{k=1}{=} \frac{3+1}{8+3\cdot1} = \frac{4}{11}$

$P(A=a_2) \overset{ML}{=} \frac{3}{8} \overset{k=1}{=} \frac{3+1}{8+3\cdot1} = \frac{4}{11}$

$P(A=a_3) \overset{ML}{=} \frac{2}{8} \overset{k=1}{=} \frac{2+1}{8+3\cdot1} = \frac{3}{11}$

prefix-free code = assign a prefix free code to the set of outcomes (we just replace heads and tails with zeros and ones). The benefit is that if we want to encode a sequence of these outcomes, we can just stick the code one after another and we won't need any delimiters. A decoder will know exactly where each codeword ends and the next begins.

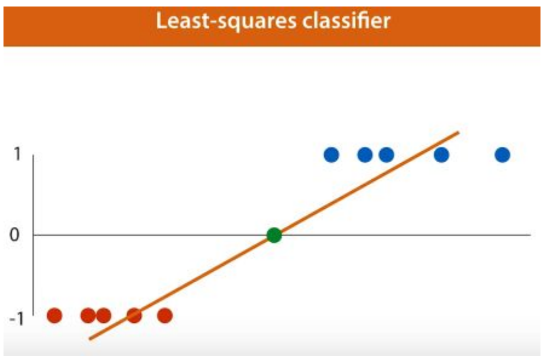


used in **Arithmetic coding** → It turns out we can model almost any distribution in such a way that the biggest difference in codelength is no larger than a bit.
The higher the probability of an outcome, the shorter its codelength.
Usually interested in have high code lengths, so one bit can be ignored. Every code gives us a distribution and every distribution gives us a code.

the Minimum Description Length Principle/ MDL(measured in bits) = A model that allows us to compress the data is a model that has learned something about the data. The better the compression, the more we've learned. Balance model complexity by storing the model, and then the data given the model. Informally states that compression and learning are strongly related.

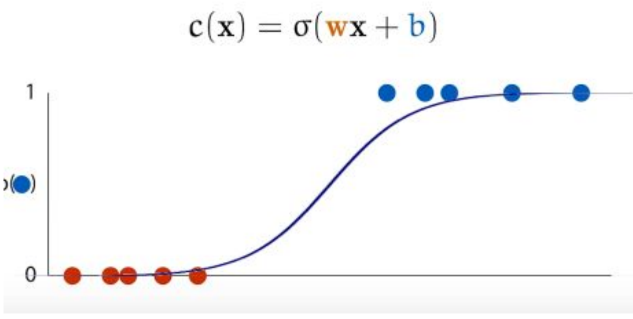
The best way to think of MDL model selection is in a sender and receiver framework. The sender is going to see some data, and is going to send it to the receives. Before observing the data, the sender and receiver are allowed to come up with any scheme they like. But afterwards, the data must be sent using the scheme, and in a way that is perfectly decodable by the receiver without further communication. We usually assume that there is some language to describe a model that the sender chooses. The sender described the model and then the data given the model. Sender → data → receiver. Minimizing the total description length is equivalent to maximizing the numerator of Bayes' rule (the denominator doesn't affect the argmax).

Entropy→ measure of uniformity (/certainty) H(p): The more uniform our distribution is (the more unsure we are) the higher the entropy. measured in codelengts, expressed in bits
Cross entropy= expected code length if we use q, but the data comes from p
p(X): the source of our data q(X): our model. minimal when p=q (and equal to the entropy).
KL divergence= expected difference in code length between p and q. Or, difference in expected code length
<u>H(p, q) and KL(p, q): Good measures of distance between model and truth.</u>



the least squares classifier= if function higher than 1, assign this class and lower than 1, assign other class. set all points higher or lower than 0, and fit the points with regression. The numbers are squared, so if you pull them twice, come back 4x. Think of them as rubber bands> pull harder further down line

However, other option: instead of giving red and blue arbitrary values, we given them probabilities: the probability of being blue, which is 1 for all blue points and red for all red points → still similar to linear classifier, still same range → *what we need, is a way to squeeze that whole range into the range [0, 1]* → **logistic sigmoid**: domain/input is the entire real number line (-∞ *till* +∞) and the range/output is [0,1].

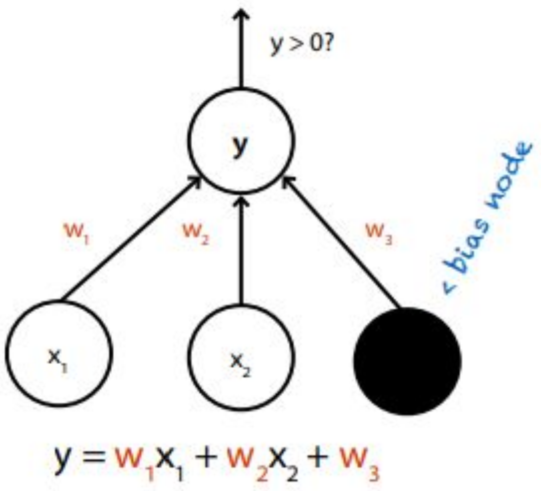


An interesting property of the logistic sigmoid is the *symmetry* given in the second line. The remainder between sigma(t) and 1, is itself a sigmoid running in the other direction. Applying the sigmoid to the output of our linear function gives us a nice probability distribution for all points in our feature space.

Use sigmoid function to turn a linear classifier into a discriminative probabilistic classifier: **Logistic regression** → Use cross entropy loss to find good values. Derive the gradient and search for good weights. No analytical solution, but the problem is convex. Sums the residuals for loss function. Points close to the decision boundary large influence, points far away almost no influence at all.

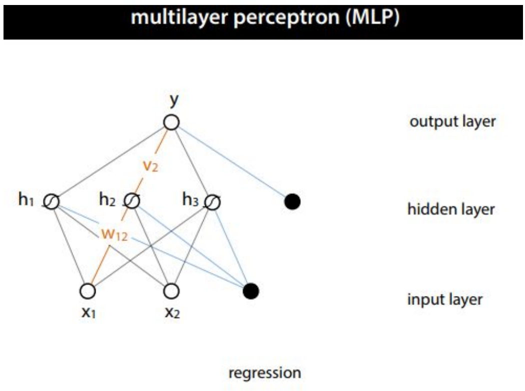
Lecture 6: Deep learning 1

"Deep learning is a new name for neural networks."

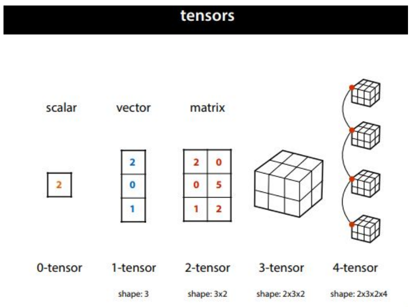


This is the **perceptron**: a simplified model of a single neuron (brain cell). It takes a number of inputs (the features), multiplies each by a separate weight and sum them. See if they are bigger or smaller than 0, then its one class, otherwise it's another.

The challenge is to choose the weights so that the perceptron classifies correctly. Note the bias node, an input node that is fixed to 1. Similar to a linear classifier. The brain contains many cells. Ultimately, we want to chain these perceptrons together, connecting this one's output to the input of another one, and build models out of multiple perceptrons → add some non-linearity (activation function/sigmoid function)



Most popular way to combine these → **multilayer perceptron.** Start with 1 layer of input units, then one layer called the hidden layer of these sigmoid outputs. Every node on the input layer is connected to every node on the hidden layer, and then we move everything to an output layer. Between layers everything connected, but within a layer no nodes are connected. Usually 1 hidden layer→ contains sigmoid activation functions.



Tensors = basic building blocks of neural networks, generalization of

what a matrix is. Data (vector 1, matrix 2), images (3) and image datasets (4) can also be expressed in tensors.
Then, we build layers with these tensors:
generic "layer"= Any differentiable function from one tensor to another tensor can be a layer.
Dense layer (MLP): vector-to-vector layer, parameterized by a weight matrix W. input * weight → output
Non linear activations (such as sigmoid) are usually defined as separate layers → called activation 'layers'

How do we choose the right weights/parameters?
Turn gradient descent into **stochastic gradient descent,** then we turn that into **backpropagation (**starts with computing the loss for the gradient and propagates the gradient/error down the network)

stochastic gradient descent → an adaption to the gradient descent, so it can be used for larger datasets. Method: just picks one data point, compute the gradient of the loss with respect to just that one data point (instead of over whole dataset). then you take a step in the opposite direction of the gradient for every data point. Move around more random. looks at small chunks of the data per iteration. also pick a loss function(MSE for regression, cross entropy for classification)
• Fast, low memory overhead
• Gradients average out over many iterations to the fulldata gradient.
• Added stochasticity helps to escape local maxima.
• batch size: tunes stochasticity as well as memory use (minibatching: compute loss over batch/small number of points)

Automatic differentiation/ autodiff/ backpropagation→ Computes the gradient for all weights in all layers. half numerically (locally, shape of function) and half symbolically (globally, numeric approximations) to find gradient. If we describe our system as a composition of modules (modular), repeated application of the chain rule turns the gradient into the product of the gradient of each module with respect to its arguments.
It distributes the error back down the neural network (or any other computation graph). > used to train neural networks

We don't derive a full symbolic expression for the gradient. But we do use the fact that we have a symbolic (differentiable) expression for the function itself. Automatic differentiation is very precise, and requires no more memory or time than the forward pass.

Forward pass: x=0 → compute the derivative with respect to a specific output. Pick X. Forward pass computes the output, **backward pass** computes your gradient. You don't need an autodiff system like tensorflow to implement backprop. You can also do it manually for a given network (backprop is much older than systems like tensorflow).

Multivariate chain rule: comes into effect when the output depends on a weight along multiple paths in the computation graph: sum them and use chain rule. For instance when

we compute the loss function over a batch of multiple data point.
Is needed when the output depends on one of the variables for which we are taking the derivative. Never take the derivative with respect to x (the input), only with respect to the weights.

rules for autodiff systems:
1. Computation graph is a DAG > directed acyclic graph (so no cycles)
2. Inputs, intermediates, weights are tensors.
3. Output is a single scalar (1 number). Computation graph contains the model and the loss.
4. Nodes (aka functions or ops) implement forward and backward.

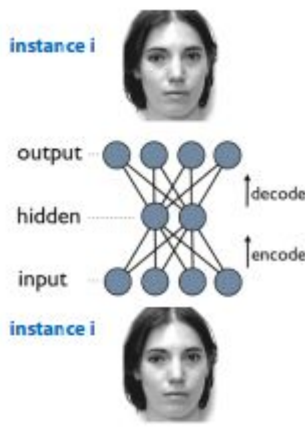
When adding layers: initial weights should keep the data centered, and the variance stable
initialization → make sure your data is normalized/whitened. Initialize your weight matrix as a random orthogonal matrix. transforms your input but only rotates it.

Activation functions
hidden: sigmoid, tanh. Doesn't always work so we often use ReLU (derivative is 0 or 1).
output: sigmoid, softmax → take the exponential and normalize it
Cross-entropy loss: loss function for multi-class classification

convolutional layers: pruning → Simplify networks in a smart way, so that you can still train it with stochastic gradient descent. We will do two things to simplify our network:
- Remove as many connections as we can
- Couple certain weights: force them to take the same value

Convolutional neural network → use the information about which pixels are together to decide which connections to throw away. Arrange the input nodes in a grid, to decide which connections to remove: we connect each node in the hidden layer just to a small nun neighbourhood in the input (here n=2). We do this for each such n x n neighbourhood in the input. Each node in the hidden layer has just 4 incoming connections. Go from one image of 5*5 to 2 images of 4*4 called **a map/filter/kernel.** If you increase number of maps, decrease resolution. *max pooling* → reduces resolution even more, but increase number of images (low resolution but with a lot of channels)

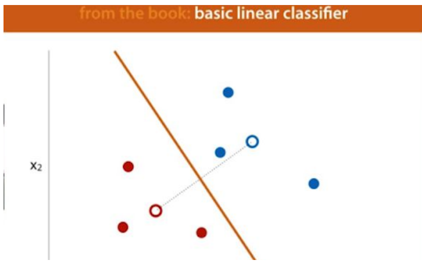
Most Convolutional networks also include *pooling layers*. These are built in the same way as the convolution layers, but they contain no weights. They pass on the largest value in the neighbourhood. The first layer gives us several convolution filters (one per map).
Takes a pixel neighbourhood and averages the pixels in it, creating a blurred result of the input. While it may seem to be throwing away valuable information, what we actually get is a representation that is i**nvariant to noise**.
auto-encoders = The input and output are the same: the networks job is to reconstruct the input. It has to do so using a layer with fewer nodes. Once the network is successfully trained, the values in the middle can serve as a low dimensional representation of the data. If we want to increase the



dimensionality of the data, this trick doesn't work. If the middle layer is bigger than the input layer, the network can just copy the input.

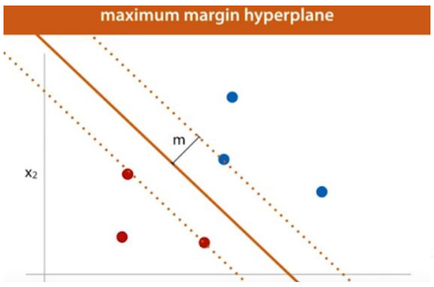
Demonising autoencoders = We can add a little noise to the input and train the network to remove the noise. This way, the middle layer becomes a representation that is helpful in removing noise.
Once trained, we can stack these auto encoders. This is the promise of **deep autoencoders**: once projected into the low-dimensional space, the instances will form natural clusters. For instance, different images of the same person would cluster together.

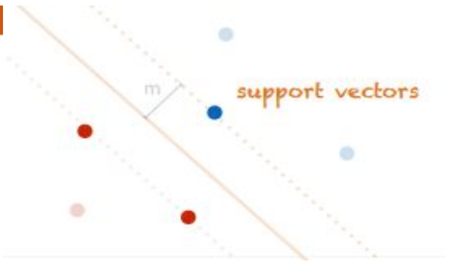
Lecture 7: Linear models 2



- wTx + b is a linear function
- wTx + b > 0? is a linear decision boundary. One boundary can be specified by infinitely many w, b (we have an extra degree of freedom).
- Accuracy/AUC is not a good loss function for search, even if it's what we're ultimately after.

Support vector machine (SVM)→ loss function. It draws a line that has the largest (equal) distance to the most nearby points (*maximum **margin** hyperplane*). It measures how far the points are from the line.
Choose the support vectors → maximize "2x the size of the margin"





Points closest to the line called the **support vectors**→ define the line. If all data would be removed except the support vectors, you can still draw the classification boundary.

How to draw this line? It's overdetermined, so we can use: w^Tx+b= -1 & 1 (line is 0)

but how do we choose the support vectors? use the most extreme points (closest to where it should be linearly separated) Pick the vectors that give the largest margin → maximize the value "2x the size of the margin" Maximze 2/||w||

'Hard margin' SVM → *no points* are allowed to violate the margin. Note how much of our objective is encoded in the constraints. Without them, we could just set w to the zero-vector and Bind a (useless) minimum trivially. ½ ||w||

‘Soft margin’ SVM → pi is a ‘slack parameter’. It relaxes the constraint a little, to allow *some points* to fall in the margin, but we pay a price in the loss function (which get higher if we allow bigger penalties. C is a hyperparameter, indicating how much we care if the margins are violated. ½ ||w|| + C(sum Pi).

→ So, the objective function has a penalty added to it, making it bigger. But, the margin we are now able to create is much wider. C lets us trade off these objectives.

A fork in the road: two options
1) Express everything in terms of w, get rid of the constraints
+ Allows gradient descent & backpropagation to be used
+ Good for use in deep learning

Gives us a loss function we can apply to any model (gives us a problem to minimize the function). For instance when training a neural network to classify, this makes a solid alternative to cross-entropy loss. This is sometimes called the L1-SVM (loss). There is also the L2-SVM loss where a square is applied to the pi function, to increase the weight of outliers

2) Express everything in terms of the support vectors, get rid of w
- Doesn’t allow error to propagate back
+ Allows the kernel trick to be applied.

Leave the constraints, but rewrite them. We will get rid of W. Rewrite everything as a constraint in terms of weights on the data → usually done with help of **Lagrange multipliers.** Rewrites function to make it easier to solve (with easier constraints). However, it’s now no longer a minimization problem, you have to solve the gradient (equal to 0). Cannot use gradient descent for this. Lagrange used dot product.

The kernel trick → If you have an algorithm which operates only on the dot products of pairs of points in your data, you can substitute the dot product for a kernel function. Calculate support vectors and decision boundary based on those dot products.

Polynomial kernel: feature space for d=2: all squares, all cross products, all single features. feature space for d=3: all cubes and squares, all thews-way and two-way cross products, all single features. ^D is a hyperparameter.
RBF kernel: feature space: infinite
Kernels in data space: text, dna, proteins: string kernels, graphs: WL distance

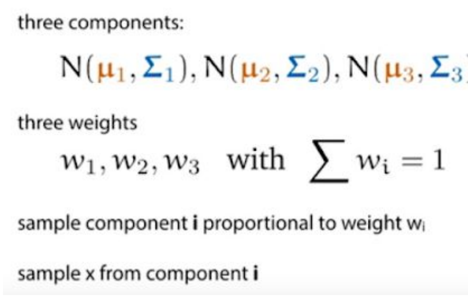
Lecture 8: Probabilistic Models 2

<u>To make a probability density function: (sums over whole domain, needs to be 1)</u>
Normal distribution needs a definite scale + be an exponentially decaying tail. A stronger decay has as benefits that it *flattens out at the peak* (giving a nice bell-shaped curve) and it has an *inflection point*: the point where the curve moves from decaying with increasing speed to decaying with decreasing speed. Use the inflection points as range.

It needs to sum to one. This is done by integrating over the whole real number line → gives us a function that sums to 1 over the whole of its domain. The expected outcomes lie inside the inflection circle. Multivariante -> more than one variable, univariate -> 1 variable

maximum likelihood principle= Reasonable criterion for fitting models. It answers what makes a good fit for probability distribution, what fits my data well? Helps in chooses the distribution to apply arg max p(x|0) good for large datasets.

Log likelihood: What we *maximise* to fit a probability model
Loss: What we *minimise* to fit a machine learning mode


three components:

$$N(\mu_1, \Sigma_1), N(\mu_2, \Sigma_2), N(\mu_3, \Sigma_3$$

three weights

$$w_1, w_2, w_3 \quad \text{with} \quad \sum w_i = 1$$

sample component **i** proportional to weight w_i

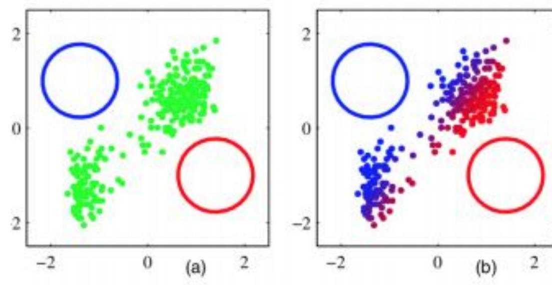
sample x from component **i**

Gaussian (= aka normal distribution) distribution mixture models → combine multiple gaussians into a single probability distribution. Attach a weight/probability to each model. Gradient descent is possible, but a closed form solution isn't. Hard to fit.

We can fit Gaussian mixture models (and other hidden variable models) with the **expectation–maximization (EM) algorithm (**similar to K-means but fancier)**=** expands on k-means by replacing the clusters with Gaussians and by allowing points to 'belong' to each Gaussian 'to some degree'. In other words, each Gaussian takes a certain *responsibility* for each point.
EM (intuitive) → initialise components randomly (pick some random MVN's + give weights) loop:
• expectation: assign soft responsibilities to each point
• maximization: fit the components to the data, weighted by responsibility

Responsibility= how big we guess the chance that a certain point belongs to a certain distribution. divide overall between 0-1, sum to 1.
Calculate = red responsibility / all responsibility.

Here we see EM in action. We start with two random Gaussians and color the points by how much responsibility each component takes. The blue points are mostly claimed by the blue component, the red points are mostly claimed by the red, and the purple points in the middle have the responsibility divided equally between the components.

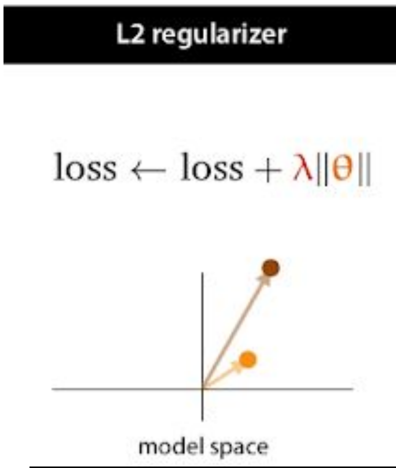


Useful for clustering (unsupervised, so have to draw own conclusion based on data): to break population up in groups & classification.

Lecture 9: Deep learning 2

loss curve = the loss curve shows you whether your model is converging, how muchness it is experiencing and whether that noise is diminishing as the model converges.
If there is a generalizing solution it always finds it, and if there isnt, it runs around until it finds the overfitting result.

what if my model starts overfitting?
1) Reduce capacity (so it has to remember less, making it simpler)→ Fewer hidden nodes, fewer layers
2) Add a regulariser (-> additions to your model that force it to prefer simpler models to more complex ones by giving them more weight): L1, L2 , Dropout



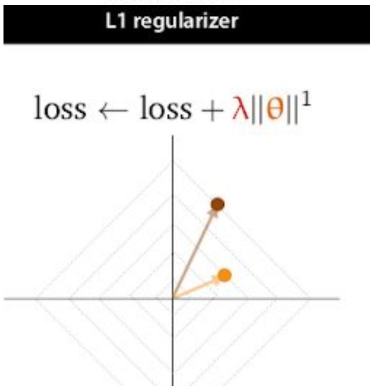
L2 regularizer: add a penalty to the loss function. Collect all your parameters into a big vector (which we'll call θ), and you add the length of this vector to the loss function (multiplied by a hyperparameter λ, which controls how strongly we want to regularize).
In this image the dark brown model will get a stronger penalty from the regulariser than the light brown one, because its parameters are further from the origin. Like tilting a bowl.

$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$

L1 regularizer: penalizes things that are not moving along the axis, **enforces sparsity.** Any parameter that gets close to an axis (i.e. close to zero) will tend to "snap" to the axis. as a result, it becomes easier to store and read. Like a square bowl.

$$||\theta|| = \sqrt{w^2 + b^2}$$

$$||\theta||^p = \sqrt[p]{w^p + b^p}$$



L^p regularization
Provides a "soft" preference for simpler models.
L2: Simpler means <u>smaller parameters.</u> Also known as ridge regression.
L1: Simpler means <u>smaller parameters and more zero parameters</u>. Also known as lasso regression.

Dropout → another regularization technique for large neural nets. During training, we simply remove hidden and input nodes (each with probability p). This prevents **co-adaptation**. Memorization often depends on multiple neurons firing together in specific combinations (turns nodes on and off randomly). Dropout prevents this.

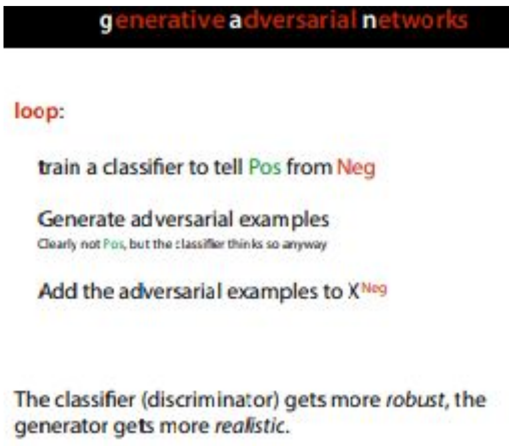
Generative modelling aims to create a model of the distribution that generated the data, in such a way that we can sample from that distribution. Two types: GANs & VAEs

A plain neural network is purely deterministic. It translates an input to an output and does the same thing every time. <u>How to we turn this into a probability distribution?</u>

1) Take the output and interpret it as the parameters of a multivariate normal. If the dimensionality of the output is large, we'll often take Σ to be a diagonal matrix. This doesn't produce very interesting distributions (it's always just an MVN), but it does allow the network to communicate how sure it is about the output: the smaller the variances in Σ, the surer it is about its output. This has the effect of smoothing the loss surface, which can help training a lot.

2) Keep the output deterministic, but sample the input from a standard MVN. Probability as input
3) Combine both: we sample the input from a standard MVN, interpret the output as another MVN, and then sample from that.

How do we train models like this? If we sample a random point from the data, and sample a point from the model and train on how close they are, we get something called **mode collapse (**=the many different modes of the data distribution end up being averaged ("collapsing") into a single point). We want the network to imagine details, *not to average over all possibilities.*



generative adversarial networks

loop:

train a classifier to tell Pos from Neg

Generate adversarial examples

Clearly not Pos, but the classifier thinks so anyway

Add the adversarial examples to X^{Neg}

The classifier (discriminator) gets more *robust*, the generator gets more *realistic*.

→ To avoid mode collapse: **"generative adversarial networks (GANs)":** high performance in image labeling. Researchers found out that by very slightly distorting the input, they could make the network output completely bad predictions.
Instead, you can add these adversarial examples to the dataset (as negatives) and retrain your network to make it more robust.

End-to-end GANs → The **generator** takes an input samples from a standard MVN and produces an image. The **discriminator** takes an images and classifies it.

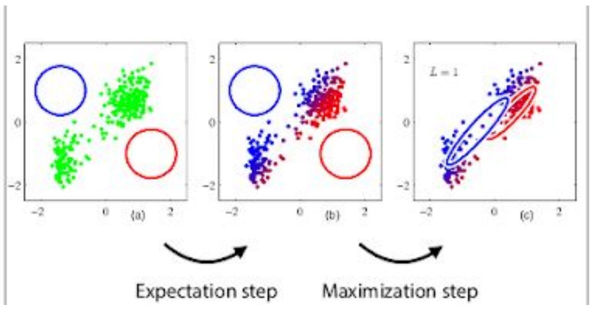
To train the discriminator, we feed it examples from the training data, and provides the training labels from the training data. We also sample points from the generator (whose weights we don't update) and label these as negative.
To train the generator, we freeze the discriminator and train the weights of the generator to produce images that cause the discriminator to label them as Positive.

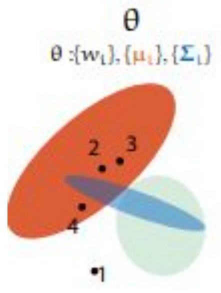
Conditional GAN used when we want to train the network to take an input, but to generate the output probabilistically. Here, the network needs to fill in realistic details. The network should pick one, and not average over all colors. Input & output have to be matched.
If we randomly match one image in one domain to an image in another domain, we get mode collapse again.

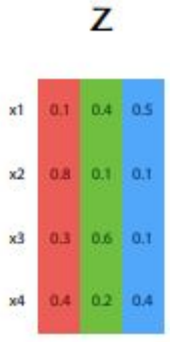
CycleGANs can handle this (+unpaired images in different domains) by adding a "cycle consistency term" to the loss function. Uses two big bags of unmatched pictures that are not mapped to each other but they do describe their domain very well. Need to transform between domains. If you randomly map the pictures in one domain in another domain you're going to get mode collapse again. So, you need additional constraints, to constrain your training process. Add a term to the loss function. After some training of the generators, the discriminators are retrained with with the original data and some generated data.
*Autoencode*r → pushes the input through a bottleneck (a small hidden layer), and tries to reconstruct the output as well as possible. Low dimensional representation.

When turning autoencoder into a generative model, can we train for maximum likelihood directly? Problems: no closed-form solution, huge sum, intractable

EM: key insight
We can optimise for θ and z together, but:
• Given some θ, we can “optimize” z
• Given z, we can optimise θ





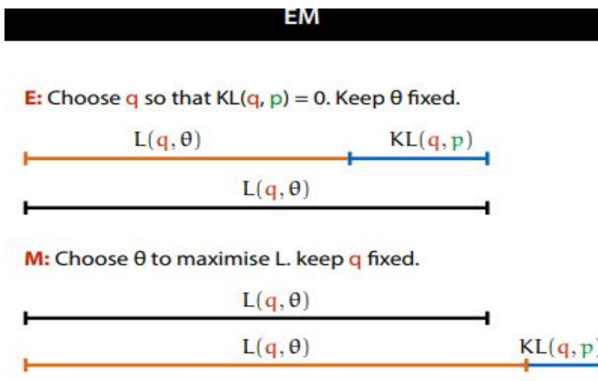


Z

x1	0.1	0.4	0.5
x2	0.8	0.1	0.1
x3	0.3	0.6	0.1
x4	0.4	0.2	0.4

EM: p(x | θ) = L(q, θ) + KL(q, p) → We call L the variational lower bound on ln p(x|θ). It’s the lower bound on what we want to maxime. Easier to maxime than p.

The EM algorithm consists of two steps. In the first we choose our q so that the KL divergence term is minimized/0. For the GMM algorithm we can compute p explicitly (these are the responsibilities), so we can set q equal to that and eliminate the KL divergence term entirely. In the second step, we choose new parameters θ, to optimise the L-term. p has now changed, to the KL term is no longer zero. Since both of these steps either keep p(x|θ) the same, or increase it, we have just proves that the EM algorithm converges (to a local minimum).

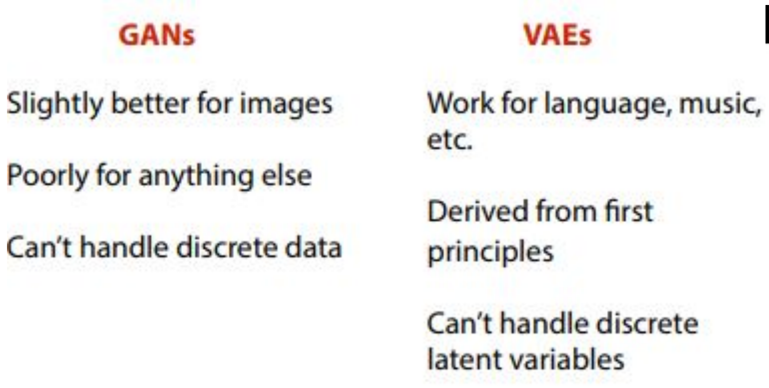


EM

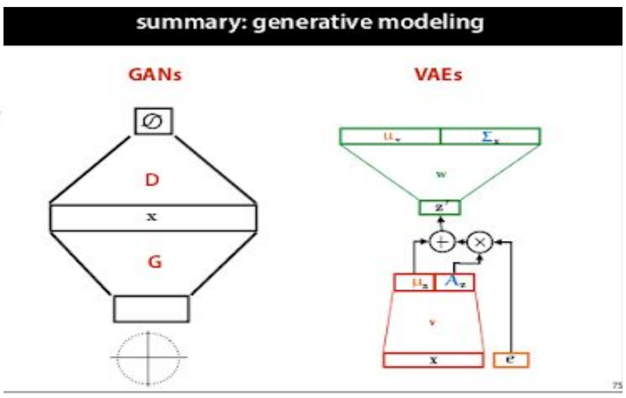
E: Choose q so that KL(q, p) = 0. Keep θ fixed.

M: Choose θ to maximise L. keep q fixed.

In the GMM case, the E step involves setting q equal to the probability of the latent variable conditional on the observed. M: Choose θ to maximise L. keep q fixed. Building a nn generative model: hard to compute p(z|θ), so we approximate p(z|x,θ)



GANs	VAEs
Slightly better for images	Work for language, music, etc.
Poorly for anything else	Derived from first principles
Can’t handle discrete data	Can’t handle discrete latent variables



summary: generative modeling

GAN: go broad, then narrow again and put the data in the middle. Mode collapse prevented by the discriminator.
VAE: we go broad, narrow , broad again and we put the data on the input and the output and we put the latent vector in the middle. Uses maximum likelihood inference
discrete= language, music, symbols etc.
derived from first principles→ directly based on probability etc.
Latent variables → middle layer. so, if your latent variables are discrete, use gan or something else. VAE better than AE.

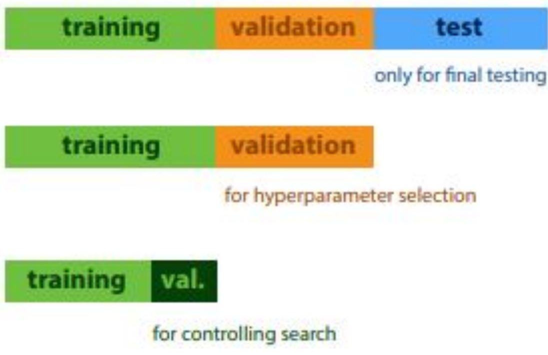
Lecture10: Tree Models (and model ensembles)

How to build a tree: standard algorithm (ID3, C45)
1. start with an empty tree
2. extend step by step (usually categorical features)
3. greedy (no backtracking)
4. choose the split that creates th*e least uniform distribution* over the class labels in the resulting segments
Note: There's no use (with categorical features) in splitting on the same feature twice

Stop conditions (= when we stop extending the tree)
- When all inputs are the same (*no more features left*) output label: majority class (=class that occurs most often)
- When all outputs are the same (*uniform class distribution* = only 1 class occurs in our segment) output label: left over class

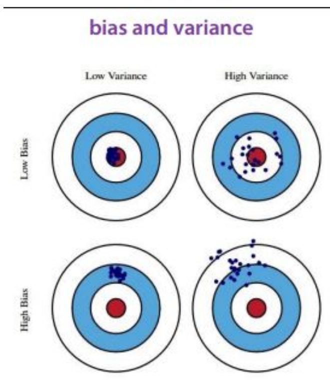
How do we define uniformity in tree classes? → entropy. Use *conditional entropy* P(A|B) → what is the chance of A when B is given.
Information gain = Io(G) = H(O) - H(O | G). Difference between the entropy before they've told G and the entropy after they told G. independent of what G is going to be, how much information am I going to get out of the fact that they've told me G. AKA, it tells us how much we reduce the entropy by picking a particular split by picking a particular feature to split tree.
Alternatives to IG: (1) minority class (aka error rate (minimize this)), (2) Gini index (aka expected error rate under random guessing)

What if all features are numeric? → turn the feature into a binary split on some threshold. Split and classify at threshold T (lower or higher than T). Trees often overfit.
For pruning: check if the set performs better on validation data with/without this node. If it performs better without node, you cut it. (On right, last split needed to prune trees).



Regression trees: for numeric features, we cannot use entropy to find uniformity. We replace entropy by variance.
You can reuse numeric features, because you can use a different threshold every time.

Decision trees are not used that often → highly unstable learners + error high variance. Solution → make a random forest/decision forest: collection of decision trees (ensemble method)

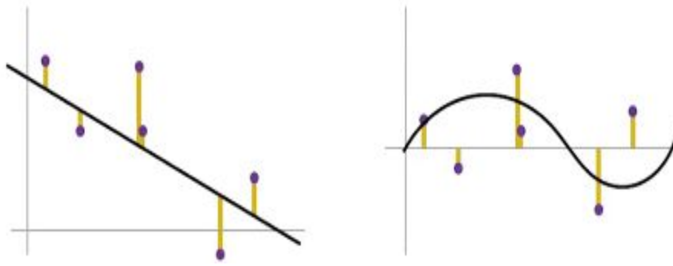


Decompose error into two problems: bias and variance. The average error of many models with high variance is low.
High variance/unstable learner = when your hit errors are very spread out, but if you average over them, you would end up with a good result.
High bias= when the results are very precise/accurate (every time), but not in the right place.

For solving high variance → bagging (bootstrap aggregating)
Resample K datasets, and train K models (this collection is our ensemble). For a new point, ask the models where to classify it in.The ensemble classifies by majority vote or probability.

For solving high bias → the hypothesis boosting question(/boosting (algorithms))
Train a sequence of classifiers and looks at instances from previous classifiers that were wrong. Increase the weights of those examples + normalize weights. Reweight the data and over the new weights, we train the classifier. The better it performs the higher its weight. For the formula, the closer the error is the one, the lower the weights gets.
→ even works for weak learners (classify just better than chance), f.e. decision stumps (decision tree with 1 node) -> cheap to train, and still get powerful classifier due to ensemble.

Gradient boosting → Subtract what your model predicted from the residuals and you get a dataset with just residuals: *fit a line through that.* Sum original line + residual line → better fitting model.



Boosting is a good way to improve performance but you shouldn't use it when comparing models, because then the models need to be looked at in isolation.

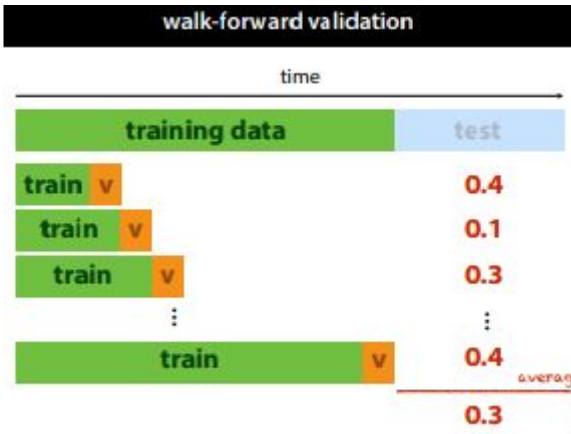
Stacking → another ensemble method. Train a sequence of models and add the models to the original data set. On these models, you train another classifier (a combiner), which combines the original features with the judgements of the ensemble and learns how to combine these different judgements. Your combiner can then judge, based on the data, decide which model is right in which case. Uses judgements of different models for different points in your feature space.
Combiner: usually on logistic regression. If NNs are used for the ensemble, the whole thing becomes one big neural network. Whereas deep learning aims to go deep, ensemble aims to go wide. Deep learning goes to new levels of abstraction etc., while ensembling cleverly segments your instance space.

Lecture 11: Models for Sequential Data

Sequences categories:
1. Numeric n-dimensional time series→ f.e. sun cycles in vectors
2. Symbolic (categorical) n-dimensional → sequence of symbols/words/music. Difficult to represent.

Distinction in sequence between:
1. Single sequence -> 1 time series. For instance, predict sales, predict traffic to website
2. Set-of-sequences -> multiple time series

Can use **feature extraction**: to predict next value (f.e. t-1 etc.) Many other features are possible: mean over the whole history, mean over the last 10 points etc. But, there's a problem: if we shuffle this data, and pick a test and training set, the classifier gets access to data from the future. Make sure the time stays true to its T → **walk-forward validation.** To check, use cross validation.



$$p(W_1 = \text{congratulations}, W_2 = \text{you}, W_3 = \text{have}, W_4 = \text{won}, W_5 = \text{a}, W_6 = \text{prize} \mid \text{spam})$$

$$\begin{aligned} & p(W_4, W_3, W_2, W_1) \\ &= p(W_4, W_3, W_2 \mid W_1)p(W_1) \\ &= p(W_4, W_3 \mid W_2, W_1)p(W_2 \mid W_1)p(W_1) \\ &= p(W_4 \mid W_3, W_2, W_1)p(W_3 \mid W_2, W_1)p(W_2 \mid W_1)p(W_1) \end{aligned}$$

$$p(\text{prize} \mid \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations})$$

^ chain rule of probability/ Joint probability = product of probabilities of each individual words in the sentence conditioned on the words that came before it.

Markov assumption → assume the probability of price depends only on the two words (or any amount of words) after it. Like naive Bayes, not true, but makes it easier.

$$p(\text{prize} \mid \text{a}, \text{won}, \text{have}, \text{you}, \text{congratulations}) = p(\text{prize} \mid \text{a}, \text{won})$$

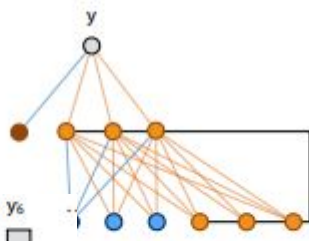
$$\underline{p(\text{prize} \mid \text{a}, \text{won})} \approx \frac{\#(\text{prize}, \text{a}, \text{won})}{\#(\text{a}, \text{won})}$$

Markov chain = combining probabilities (shakespeare example)

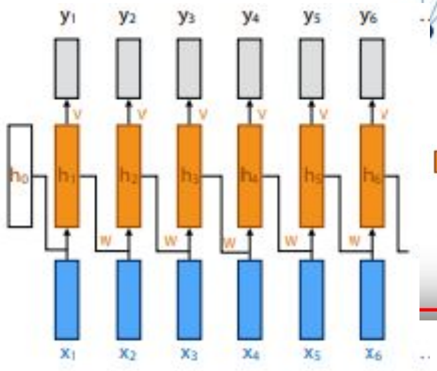
• 0-order Markov model: Naive Bayes. For spam classification higher order don't improve performance. For other tasks they do.
• Short documents are vastly more likely than long ones Doesn't matter for classification. In other settings, conditioning on length may be necessary.
• Laplace smoothing, Same as before: adapt the estimator by adding pseudo-observations.

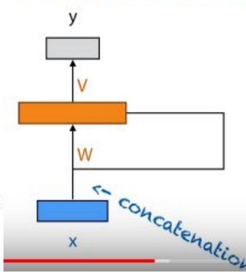
Embeddings show similarities in words, so words with similar meanings can be identified. Based on **distributional hypothesis**, which states that words that occur in the same context often have similar meanings → This means that if we have some word X, and we create a probability distribution P modelling representing how likely a word is to occur in the context of X, we can use that distribution P as a proxy for the meaning of X.

Encode our words in **1-hot vectors** (zero everywhere except 1 point) vector is length of vocabulary. **Word2Vec (skipgram version)** → creates embedding vectors for words. A bit like an autoencoder, except we're not reconstructing the output, but the *context*. <u>The input is linear, output is softmax (ensures all outputs sum to 1).</u> Predicts a probability distribution on context words.
After training, throw away the top layer, and only use the bottom part (as an encoder). In the embedding space, distances and directions reflect semantic meaning.

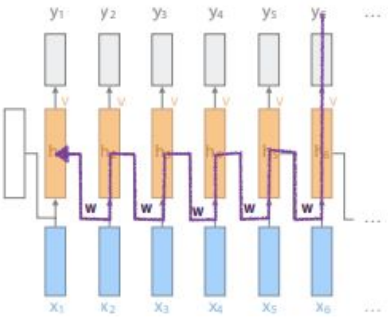


Recurrent neural networks = to represent symbolic sequences. Uses cycles. There are some connections going back to some extra input nodes, so they feed data back instead of forward. Simple small example: Concatenation= sticking it together. Feed it a timeseries.

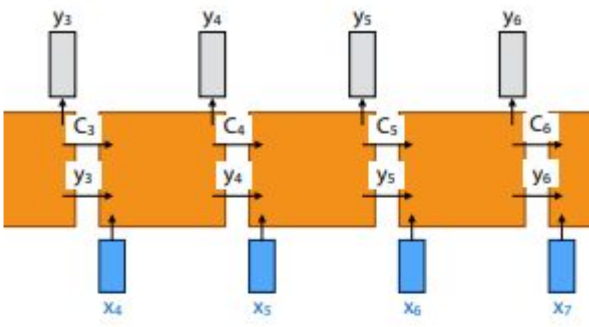




How to determine the weights/train the RNN? → **'unrolling'.** Now the whole network is just one big, complicated feedforward net. Note that we have a lot of shared weights, but we know how to deal with those. The hidden layer is initialised to the zero vector.
< Backpropagation through time



The discussed NN have a problem with long term memory processes → can't have a long term memory for everything. You need to be *selective,* and you need to learn to select word to be stored for the long term when you first see them. **Long short term memory (LSTM) =** Selective forgetting and remembering further back, controlled by learnable 'gates'. Passen on: C= memory (vector), Y= output from previous step.
Can use LSTM as a language model to predict distributions/the next word in a sequence/characters.



Tanh= sigmoid but more stretched out so the smallest value is -1.
Sampling: Don't unroll the network. Feed the network a small sequence of characters from the data as a seed.
Can use LSTM in combination with a VAE to go from sequence to sequence. Have encoder

that maps data to latent vector and decoder that maps your latent vector to an output. Also used for automatic captioning.

Final notes:

- For word-level LSTMs: use embeddings on the input later (either pre-trained w2v embeddings, or an embedding layer that is trained end-to-end)
- Sequence modeling: we can use existing methods through feature extraction, but be careful about validation
- markov models: simple, finite-memory sequence modeling (classification or generation) useful models for sequences or w2v
- Word2Vec: word embeddings reflecting similarity, semantics
- LSTM: incredibly powerful language models, tricky to train

Lecture 12: Matrix models and recommender systems

Explicit feedback system: asks users for rating (1-5 stars)
Implicit information: page views, wishlist, similarities between users/movies etc.
Side information (features): movies (length, genre, actors), users (country, bio, language)

Task: Given many users and items, in combination with (in)complete implicit/explicit/side information: *predict how users u would rate item i*

To solve this → **matrix models/factorization. R=U^T M general linear model.** Is a very simple problem if we know M → Solve using least-squares optimization.

Two problems: 1. we have *a lot* of missing values in R (resulting matrix), cannot just enter 0 everywhere 2. We don't want to design and fill in M (movies) by hand, might be too diverse.

To solve problem 1: *Optimize only for the known ratings,* so you sum over all the ratings you know, you take the rating that is known. Can use stochastic gradient descent.

Negative sampling → If we only have positive ratings, we have two options:
1. Ensure that U^t M always represent probabilities (expensive, lots of normalizing)
2. Sample random movie user pairs as negative training samples
Alternating least squares → alternative to gradient descent. Optimize until it converges.
If we know M, computing U is easy & vice versa. Instead of withholding users/movies, withhold ratings.

Shortcomings of the system:
User bias: not all users are the same: For a person who always gives 5 stars, 3 stars is really bad. Have to take into account how likely it's used to give a certain rating.
Movie bias: even if you hate all romance movies, still might want to see titanic, because globally popular and many prices.
→ add biases to the formula.

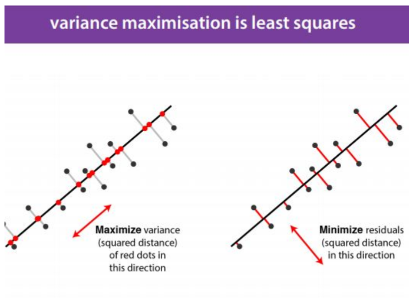
Regularization (L2): soft way of saying if the features are mutually exclusive then the vector should sum to one. Can also add it to the loss function

Cold start problem → problem in recommender systems: can't recommend stuff to new users since they don't have a history + can't recommend new movies since they haven't been rated. Need some initial recommendations. To solve:
→ include implicit 'likes'/info (watched but not rated, browsed movies etc. can be normalized)
→ include side information (compare to similar users. User features: age, origin, login etc.)

Important: time/temporal dynamics
1) The way netflix asked for ratings changed, big increase in liking → needs correction
2) Older movies often ranked higher; people look specifically for them so high rating.

Validation: No training on data from the future! All training data should come before all validation data, which should come before all test data.

Different way to think about PCA→ it reduces the dimensionality of a matrix. For each instance, it computes a smaller feature vector with fewer dimensions than the dimensions dataset such that as much information as possible is contained. Choose component (number):just before the inflection point (big drop in graph). Idea of PCA is to choose the component to maximize variance (similar to minimizing residuals) Matrix factorization with squared error loss actually the same as PCA variance reduction. Difference: in PCA explicit constraint that all components should all be orthogonal to each other/eigenvectors.


variance maximisation is least squares
Maximize variance (squared distance) of red dots in this direction
Minimize residuals (squared distance) in this direction

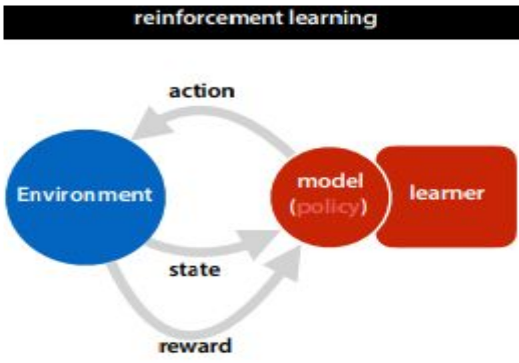
PCA: variance maximization is least squares. Objective of maximizing variance is equivalent to minimizing residuals.

- when your task consists of linking one large set of things to another large set of things, based on sparse examples, an little intrinsic information, matrix factorization may be appropriate (two big finite sets of things and a couple of links)
- remember to validate/test correctly
- extend your models with biases, regularizers, implicit likes, side information and temporal dynamics
- pca is like matrix factorization. pca usually operates on complete matrices (data matrix), MF usually on incomplete matrices (user rating etc., missing values)
- **collaborative filtering** → another name for idea of recommender systems using matrix factorization existing ratings.

Lecture 13: Reinforcement Learning

RL → agents that behave and learn in the world at the same time. <u>Online system that learns as it interacts with the world.</u>

Markov decision process = for a given state, the optimal policy may not depend on the states that came before. Only the information in the current state counts.



RL can be used (f.e.) for modeling two player games which move in succession, such as chess. All states are 0, except when you win, then it's 1. Draw is also 0. One benefit of RL is that a single system can be developed for many different tasks, as long as the interface between the world and the learner stays the same. During this model, we make the **markov assumption:** the optimal action in the current state doesn't depend on anything but <u>the current state.</u>

Reward function: r(s,a)= 1 (higher the better). In most cases, the agent will not have access to the reward function and it will have to learn them. *Deterministic policy* → every state is always followed by the same action. *Probabilistic policy* → all actions are possible, but certain ones have a higher probability. *Policy network* → learns the function from state to action (the policy) directly.

Credit assignment problem/delayed reward problem → The most recent action may not have anything to do with the received reward; might have been from long time ago and it's only now paying off.

To avoid this problem/ 3 approaches to RL:

1) **Random search** → pick a random point, and another point close to it. We compute the reward for both of them by running some simulations, and average over them. If it's better than the old points, we switch to the new setting.

2) **Policy gradients** → When reaching a reward state, label all preceding state action pairs with the final outcome. If some of these actions were bad, on average they will occur more often in sequences ending with a negative reward, and on average they will be labeled bad more often than good. Uses backpropagation.

3) **Q-learning** → compares the expected utility of the available actions (for a given state) without requiring a model of the environment. The **discounted reward** is the value we will try to <u>maximize:</u> helps to find the policy that gives us the greatest reward for all states. The **value function** gives us the discounted reward for the future: how much we expect to gain. Using the value function, define **optimal policy**, π*: policy that gives us the highest value function for all states. **Optimal value function** corresponds to the optimal policy. Can solve by rewriting or iterations. Q-learning does not tell you how to choose the action. It may be tempting to use your current policy to choose the action, but that may lead you repeat early successes without learning much about the world.

Exploration vs. exploitation tradeoff → If you find something good early on, you will over exploit that, and not explore more. Manually mix explore and exploit.

AlphaGo
- difficult because of high branching factor + reward comes very late
- the rules of Go were hardcoded into it, and it learned by simply observing existing matches
- before AlphaGo, nobody was using ML to solve these problems in games
- Minimax useless for Go → too many options/nodes
- The functions that AlphaGo learns are convolutional networks. One type is a **policy network** (from states to moves) and one is a **value network** (from states to a numeric value).
- The networks are trained by reinforcement learning using policy gradient descent. During actual play, AlphaGO uses an MCTS algorithm.

What also helped in Go was *rollouts*. Pick a node and go down quickly to a state where you either win or lose. Do a few times and move to the state where the most rollouts won. Considered to be a sort of value estimate for your function.

Finally, there was **Monte-carlo tree search (MCTS)** → combines rollouts with incomplete tree search.
1) Selection: select unexpanded node. The random walk should favour nodes with a high value, but also explore the nodes with low values (exploration/exploitation tradeoff)
2) Expansion: Once we hit a leaf (an unexpanded node), we expand it
3) Simulation: From each expanded child we do a rollout.
4) Back propagation: Updating all the nodes above this child by increasing the number of rollouts.