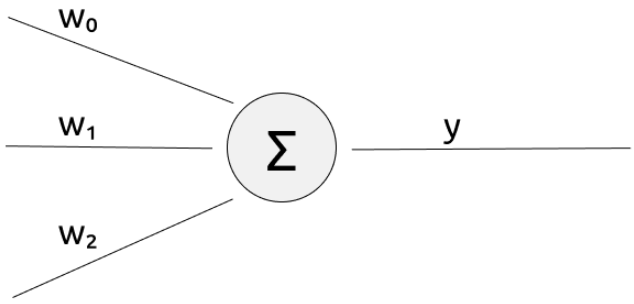
Machine Learning (module 1 part 2)

Based on Passerini's **lectures** of 2020/2021 and **pdf** of 2021/2022.

> Davide Modolo 2022



Linear Discriminant Functions

Discriminative vs Generative

Generative assumes knowledge of the distribution of the data (you want to model the distribution governing the data).

Discriminative focuses on modelling the discriminant function (the relation input -> output). You can ignore modelling the probability, focussing only on predicting the boundary

Pros and Cons of Discriminative Learning

Pros:

- with complex data, modelling their distribution can be difficult/expensive
- if the discrimination is the goal, data distribution modelling is not needed
- build the model to fitting the parameters on the desired goal (for example in image recognition, if we want to understand what an object is we can ignore the probability of pixels)

Cons:

- models less flexible because they have a fixed input and a fixed output
- for arbitrary inference task and for sampling examples (generate new x) you need generative models

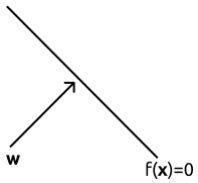
Linear Function

$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$, where w_0 is the *bias* (or *threshold*) and produces a real value

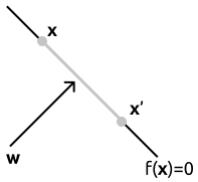
Binary Classification with a Linear Function

You want to take the produced real value to predict the class, taking for example the sign of the result: $f(\mathbf{x}) = sign(\mathbf{w}^T\mathbf{x} + w_0)$ [positive = class1, negative = class2]

The decision boundary $f(\mathbf{x}) = 0$ is an *hyperplane* H; the *weight vector* $\mathbf{w}$ is orthogonal to the decision hyperplane.



So, if we take $\mathbf{x}$ and $\mathbf{x}'$: $\forall \mathbf{x}, \mathbf{x}' : f(\mathbf{x}) = f(\mathbf{x}) = 0$ we substitute $\mathbf{x}$ with $\mathbf{w}^T\mathbf{x} + w_0$ so we get $\mathbf{w}^T\mathbf{x} + w_0 - \mathbf{w}^T\mathbf{x}' + w_0 = 0$, in the end we get that the vector $\mathbf{w}$ is orthogonal because $\mathbf{w}^T(\mathbf{x} - \mathbf{x}') = 0$ (it means that the vector $\mathbf{w}$ is orthogonal to the "grey" vector because their dot product is 0)



The Margin (confidence of classifier)

Functional Margin

The value of $f(\mathbf{x})$ is the **functional margin** (it's the *confidence* because it's the margin before predicting the following class).

Geometric Margin

The **geometric margin** (the *distance* from the hyperplane) and it's the functional margin divided by the *norm of the weight vector* (it's a normalized version of the functional margin): $r^{\mathbf{x}} = \frac{f(\mathbf{x})}{||\mathbf{w}||}$; the distance from the origin to the hyperplane is $r^0 = \frac{f(0)}{\|\mathbf{w}\|} = \frac{w_0}{\|\mathbf{w}\|}$

$\rightarrow$ a point can be expressed by its projection on H plus its distance from H times the unit vector in that direction $\mathbf{x} = \mathbf{x}^p + r^x \frac{\mathbf{w}}{\|\mathbf{w}\|}$

$$
\begin{aligned}
f(\mathbf{x}^p) &= 0 \\
\Rightarrow f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + w_0 \\
&= \mathbf{w}^T(\mathbf{x}^p + r^x\frac{\mathbf{w}}{\|\mathbf{w}\|}) + w_0 \\
&= \overbrace{\mathbf{w}^T\mathbf{x}^p + w_0}^{f(\mathbf{x}^p)=0} + r^x\mathbf{w}^T\frac{\mathbf{w}}{\|\mathbf{w}\|} \\
&= r^x\|\mathbf{w}\| \\
\Rightarrow \frac{f(\mathbf{x})}{\|\mathbf{w}\|} &= r^x
\end{aligned}
$$

Perceptron

> the starting point for neural networks

Inspired by the way the human brain processes the information, the **perceptron** (single neuron) architecture is the linear combination of input features with a threshold activation function: $f(x) = sign(\mathbf{w}^T\mathbf{x} + w_0)$

A single linear classifier can represent any linear separable concept like AND, OR, $NAND$ and NOT, but not things like XOR directly: we need to write it as a combination of what it's possible $\rightarrow (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$ so we use a networks of two layers.

In general, any boolean formula can be written as *DNF* (sum of products) or *CNF* (product of sums) with two layers of perceptrons (with a number of them exponential in the size of the input).

Learning

To make easier for calculations, we treat the bias as an additional input 1 (feature) $\hat{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$ with its weight resulting in an augmented vector $\hat{\mathbf{w}} = \begin{pmatrix} w_0 \\ \mathbf{w} \end{pmatrix}$ but we will continue writing $\mathbf{x}$ and $\mathbf{w}$.

Parameter Learning

Error minimization: *loss function* to optimize $E(\mathbf{w}, D) = \sum_{(\mathbf{x},y)\in D} l(y, f(\mathbf{x}))$ that takes the *true output* y and the *predicted output* $f(\mathbf{x})$ and computes the loss l you "pay" if you predict $f(\mathbf{x})$ instead of y. D is the *dataset*.

If you minimize the error in the training set there is the problem of *overfitting*.

Gradient Descent

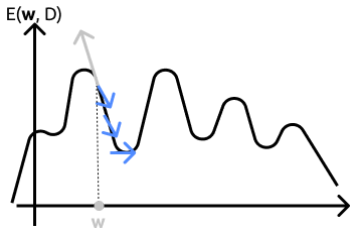
To minimize the function, you take the **gradient** and you *initialize* it and you try to "zero" it.

> the **gradient** is the slope

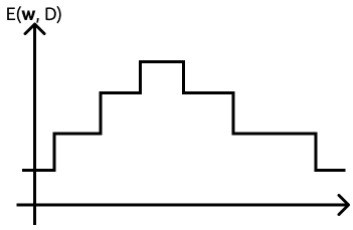
Then you iterate until the gradient is approximately zero with $\mathbf{w} = \mathbf{w} - \eta \nabla E(\mathbf{w}, D)$

The *direction* of the gradient sends towards the maximum of the function, so we go the opposite direction and repeat until the gradient is zero and we reached a **minimum** (local).

To understand how much we have to move at each step (because the gradient gives the direction), we have the **learning rate** η (too low η implies slow convergence, too big η makes us miss the minimum; there exist techniques to adaptively modify η).



Using the number of miss-classified samples (training loss) as the performance measure result in (due to the fact that the loss value is an integer that represent the number of errors):



making the grading descent impossible to use (the grading descent needs a smooth error function).

To resolve this problem we use the confidence loss, higher the confidence in the wrong prediction, higher the result in the function: $E(\mathbf{w}, D) = \sum_{(\mathbf{x},y)\in D} -yf(\mathbf{x})$.

So if y is positive and $f(\mathbf{x})$ is also positive, the product is positive; if they are both negative the result is positive but if there is an error in the classification the result is negative.

It's a smooth function because $f(\mathbf{x})$ is a real value; the gradient of this error is:
$\nabla E(\mathbf{w}, D) = \sum_{(\mathbf{x},y)\in D_E} -y\mathbf{x}$

so the update of $\mathbf{w}$ at each iteration is $-\eta \nabla E(\mathbf{w}, D) = \eta \sum_{(\mathbf{x},y)\in D_E} -y\mathbf{x}$ until we reach a zero gradient.

> ∇ = gradient

Stochastic training rule

An alternative to the gradient descent, that implies fast updates because with gradient we have to scan the whole training set (because the training error is the sum of them).

We look at just one example, if it's correct we go to the next sample. When we meet a wrongly classified sample we immediately update the $\mathbf{w}$: $\mathbf{w} \leftarrow \mathbf{w} + \eta y\mathbf{x}$

Perceptron regression exact solution

Let X be a matrix $\in \mathbb{R}^n \times \mathbb{R}^d$ with n rows, one for each sample ($n = |D|$), and d columns, one for each feature ($d = |\mathbf{x}|$).

Let $\mathbf{y} \in \mathbb{R}^n$ be the output set.

Linear regression: $X\mathbf{w} = \mathbf{y}$, solved as $\mathbf{w} = X^{-1}\mathbf{y}$

But it doesn't work because X is rectangular, the system of equations is overdetermined and usually an exact solution doesn't exist.

> overdetermined = more equations than unknown.

To do error minimization we use the mean squared error (the squared error is the difference between y and $f(\mathbf{x})$): $E(\mathbf{w}, D) = \sum_{(\mathbf{x},y)\in D} (y - f(\mathbf{x}))^2 = (\mathbf{y} - X\mathbf{w})^T(\mathbf{y} - X\mathbf{w})$

where a closed form solution exists.

We then take the gradient of the error:

$$\begin{aligned}
\nabla E(\mathbf{w}, D) &= \nabla(\mathbf{y} - X\mathbf{w})^T(\mathbf{y} - X\mathbf{w}) \\
[\text{we zero it}] \ &= 2(\mathbf{y} - X\mathbf{w})^T(-X) = 0 \\
&= -2\mathbf{y}^T X + 2\mathbf{w}^T X^T X = 0 \\
[\text{we transpose everything in order to have } \mathbf{w} \text{ as column}] \ \mathbf{w}^T X^T X &= \mathbf{y}^T X \\
X^T X\mathbf{w} &= X^T\mathbf{y} \\
\mathbf{w} &= (X^T X)^{-1} X^T \mathbf{y}
\end{aligned}$$

The $(X^T X)^{-1} X^T$ is called **left-inverse** and exits if $(X^T X) \in \mathbb{R}^{d\times d}$ ($X^T X$ full rank/square).

> linear regression solved as $\mathbf{w} = X^{-1}\mathbf{y}$, here X^{-1} = left inverse

"Linear" Multiclass Classification

One vs All

One binary classification with the boundary between a class and every other class samples.

So you have m decision hyperplanes (one per class) and when you have to make a prediction, you test on all m binary classifier resulting in a confidence for each class and you take the highest confidence.

Boundaries are pieces of hyperplanes, for example between class i and class j is where $f_i(\mathbf{x}) = f_j(\mathbf{x}) \rightarrow \mathbf{w}_i^T\mathbf{x} = \mathbf{w}_j^T\mathbf{x} \rightarrow (\mathbf{w}_i - \mathbf{w}_j)^T = 0$

All pairs

One binary classifier for each class pair.

To predict a new example we see which class wins the highest number of "duels".

One-vs-all vs All-pairs

In *one vs all* you have to train m classifiers, each with all examples. In *all pairs* you have to train $\frac{m(m-1)}{2}$ classifiers (order of m^2 classifiers), but each one only with the examples of the two classes we are training on.

If the complexity of the training procedure is higher than quadratic in the number of examples, *all pairs* is faster.

Generative Linear Classifiers

What type of *generative models* produce *linear classifiers* with the function they use to discriminate?

Gaussian classifier: linear decision boundary are obtained when covariance is shared among classes $\Sigma_i = \Sigma$ (or better, it's log-linear).

Naïve Bayes classifier: $f_i(\mathbf{x}) = P(\mathbf{x}|y_i)P(y_i)$

$$= \prod_{j=1}^{|\mathbf{x}|} \prod_{k=1}^{K} \theta_{ky_i}^{z_k}(x[j]) \frac{|D_i|}{|D|} = \prod_{k=1}^{K} \theta_{ky_i}^{N_{k\mathbf{x}}} \frac{|D_i|}{|D|}$$

where $|D_i|$ is the dimension of the training set for the class i and $N_{k\mathbf{x}}$ is the number of times the feature k appears in $\mathbf{x}$; for example in a text $\mathbf{x}$, N is the number of times the word k appears.

> θ is the parameter (?)

This is useful because if I take the $\log$ it becomes $\log f_i(\mathbf{x}) = \overbrace{\sum_{k=1}^{K} N_{k\mathbf{x}} \log \theta_{ky_i}}^{\mathbf{w}^T\mathbf{x}'} + \overbrace{\log\left(\frac{|D_i|}{|D|}\right)}^{w_o}$

with:

- $\mathbf{x}' = [N_{1\mathbf{x}} \dots N_{K\mathbf{x}}]^T$
- $\mathbf{w} = [\log \theta_{1y_i} \dots \log \theta_{Ky_i}]^T$
- w_0 is the bias (since it doesn't contain any $\mathbf{x}$)

The Naïve Bayes is a *log-linear* model (as Gaussian with shared Σ)

Support Vector Machine

Properties:

- **Large Margin Classifiers**: because when they learn they separate hyperplanes (boundary between classes) and also seprating the classes with the largest margin.
- the solution (decision boundaries) can be expressed in few training example called support vectors
- the error is based on margin
- can be extended to nonlinear separation with kernel machines

Maximum margin classifiers

definition of **margin**: given the set D, a classifier **confidence margin** is: $\rho = \min_{(\mathbf{x},y)\in D} yf(\mathbf{x})$ [the minimal confidence of the classifier in a correct prediction]

> $yf(\mathbf{x})$ is the confidence in the correct prediction
>
> $f(\mathbf{x})$ is the confidence in the prediction

A classifier **geometric margin** is: $\frac{\rho}{||\mathbf{w}||} = \min_{(\mathbf{x},y)\in D} \frac{yf(\mathbf{x})}{||\mathbf{w}||}$

The margin in SVM is the distance between the closest points to the hyperplane.

An hyperplane in general is represented by $\mathbf{w}^T\mathbf{x} + w_0 = 0$, with $\overrightarrow{\mathbf{w}}$ and w_0 are the parameters, and there is an infinite number of equivalent formulation $\forall \alpha \neq 0 \rightarrow \alpha(\mathbf{w}^T\mathbf{x} + w_0) = 0$, so we have the same numbers that "zero" it.

So the **canonical hyperplane** is the one having confidence margin $\rho = 1$: $\rho = \min_{(\mathbf{x},y)\in D} yf(\mathbf{x}) = 1$ and its geometric margin is $\frac{\rho}{||\mathbf{w}||} = \frac{1}{||\mathbf{w}||}$

Hard Margin SVM

Margin Error Theorem: $\nu + \sqrt{\frac{c}{m}(\frac{R^2 \bigwedge^2}{\rho^2}\ln^2 m + \ln\left(\frac{1}{\delta}\right))}$

The probability of test error so depends on:

- ν is number of margin errors (samples that are outside the confidence margin, correcly classified samples with low confidence)
- m training example in the $\sqrt{\frac{\ln^2 m}{m}}$ so the result goes down if m goes up
- R is the radius of the space containing all the samples
- larger the margin ρ, the smaller test error (so we want the margin $\frac{2}{||\mathbf{w}||}$ to be large)

  if ρ is fixed to 1, maximizing margin corresponds to minimizing $||\mathbf{w}||$
- c is a constant

> it makes an upper bound of the generalization error (?)

The name **hard margin** is because we require all examples to be at confidence margin at least one.

Learning Problem of SVM

we want to minimize the error, so maximizing $\frac{2}{\|\mathbf{w}\|}$ means minimizing $\frac{\|\mathbf{w}\|}{2}$ and since the square root is monotonic we have to minimize $\frac{\|\mathbf{w}\|^2}{2} = \frac{\mathbf{w}^T\mathbf{w}}{2}$, so we are interested in finding the minimum $(\overrightarrow{\mathbf{w}}, w_0)$.

We can write it as the confidence margin should be grater than one $y_i f(\mathbf{x}_i) \geq 1$
$\rightarrow y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1,\ \forall(\mathbf{x}_i, y_i) \in D$

Constrained Optimization

When you do the minimization you have to keep in account the *constraints*

Karush-Kuhn-Tucker approach

A constrained optimization problem can be addressed by converting it into an *unconstrained problem* with the same solution.

e.g. we have to minimize z in $f(z)$ with the constraints $g_i(z) \geq 0,\ \forall i$

We introduce a variable $\alpha_i \geq 0$ called Lagrange multiplier for each constraint i and we rewrite the problem as Lagrangian: $\min_z \max_{\alpha \geq 0} f(z) - \sum_i \alpha_i g_i(z)$ so it is unconstrained.

We now want to minimize in respect to z and maximize in respect to α.

The optimal solutions z^* of this problem are also optimal solution for the starting constrained problem.

Now suppose we find a solution z':

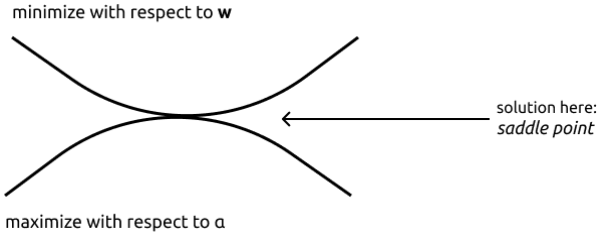
- if at least one constraint is not satisfied ($\exists i \mid g_i(z') < 0$), maximizing over α_i leads to an infinite value;
- if all constraints are satisfied, maximizing over α sets all elements in the sum to zero so that z' is a solution for $\min_z f(z)$.

If we apply the KKT approach to SVM we have:

$$\min_{\mathbf{w}, w_o} \quad \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to: } y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1,\ \forall(\mathbf{x}_i, y_i) \in D$$

> $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1$ is the $g_i(z) \geq 0$ of the previous example

This **Lagrangian** is minimized with respect to $\mathbf{w}$ and maximized with respect to α



$$L(\mathbf{w}, w_0, \alpha) = \frac{\|\mathbf{w}\|^2}{2} - \sum_{i=1}^{m} \alpha_i (y_i(\mathbf{w}^T\mathbf{x}_i + w_0) - 1)$$

now we want to minimize with respect to $\mathbf{w}, w_0$ and maximize with respect to α

So we take $\nabla_{\mathbf{w}} L = \nabla_{\mathbf{w}} \frac{\mathbf{w}^T\mathbf{w}}{2} - \nabla_{\mathbf{w}} \sum_i \alpha_i y_i \mathbf{w}^T\mathbf{x} = \frac{\not{2}\mathbf{w}}{\not{2}} - \sum_i \alpha_i y_i \mathbf{x}$

and now we set $\mathbf{w} - \sum_i \alpha_i y_i \mathbf{x} = 0$ getting $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}$, but this is not the solution because $\mathbf{w}$ is defined in terms of α

So now we make the derivative $\frac{\delta L}{\delta w_0} = \frac{\delta(-\sum_i \alpha_i y_i w_0)}{\delta w_0} = -\sum_i \alpha_i y_i = 0 \rightarrow \sum_i \alpha_i y_i = 0$

> For the full process: **svm 2** lecture 34 minutes in

The result of this **dual formulation** is $\sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m \alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j = L(\alpha)$ so we have:

$$\max_{\alpha\in\mathbb{R}^m} \quad \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m \alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j \text{ subject to } \alpha_i \geq 0 \quad i = 1,\dots,m \quad \sum_{i=1}^m \alpha_i y_i = 0$$

The **primal variables** are the one in the original problem, this is dual formulation because it contains the α that are the **dual variables** introduced with the *Lagrangian*.

The dual is still a quadratic problem due to the squared α.

The result is that $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ can be written both in form of the primal and of the dual because we know that w is equal to $\mathbf{w} = \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i$

The decision $f(\mathbf{x})$ (defined as **decision function**) on $\mathbf{x}$ is basically taken as a linear combination of dot products between training points and $\mathbf{x}$, so if $\mathbf{x}_i$ is similar to $\mathbf{x}$ it will have a high dot product: each training sample pulls towards its class with a weight that depends on the dot product

In the optimal solution, each component of the lagrange multiplier $\sum_{i=1}^m \alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + w_0) - 1)$ should be $= 0$, either:

- $\alpha_i = 0$, so the example $\mathbf{x}_i$ does not contribute to the final solution
- $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) = 1$, so the confidence for the example should be 1

The *support vectors* are where $\alpha_1 > 0$ that lays in the hyperplanes where *confidence* $f(\mathbf{x}) = 1$

Decision function bias

We can compute the bias w_0 by taking any sample $\mathbf{x}_i$ where $\alpha > 0$ (so any *support vector*), putting it in $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) = 1$ and resolving for $w_0 = \frac{1 - y_i\mathbf{w}^T\mathbf{x}_i}{y_i}$

Soft Margin SVM

There is a *problem* with hard margin SVM: if two examples are too close, the *margin* could be too small (and this happens even if there is only one example that moves the decision hyperplane too close), resulting in **overfitting**.

Slack variables

To avoid this problem, we go with **soft margin SVM**: we want to maximize the margin and we want all examples to be on their correct side, but this constriction can be *relaxed* (if there are few examples that are not in the correct side, and ignoring them leads to a higher margin, we can accept it, because is better than overfitting). Formalized as:

$$\min_{\mathbf{w}\in X, w_0\in\mathbb{R}, \xi\in\mathbb{R}^m} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m}\xi_i$$
$$\text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 - \xi_i, \quad i = 1, \dots, m$$
$$\xi_i \geq 0, \quad i = 1, \dots, m$$

Where:

- ξ_i is the **slack variable**, it represents the *penalty* for the sample $\mathbf{x}_i$ not satisfying the margin constraint $\rightarrow$ larger the ξ, further you are from satisfying the constraints
- $C \geq 0$ is the **regularization parameter** that trades-off *data fitting* and *size of the margin* $\rightarrow$ smaller the C more exceptions you can accommodate (you have to decide C before doing the learning task, it's an hyperparameter)
- the sum of the slacks is minimized together to the inverse margin

> remember: $\min_{\mathbf{w}, w_o} \quad \frac{1}{2}\|\mathbf{w}\|^2 \quad$ subject to: $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1$ part is the hard margin SVM we want to soften

Regularization Theory

Generalization of the concept written before as: $\min_{\mathbf{w}\in X, w_0\in\mathbb{R}, \xi\in\mathbb{R}^m} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} l(y_i, f(\mathbf{x}_i))$

It combines a **complexity term** $\frac{1}{2}\|\mathbf{w}\|^2$ (before it was the *margin*) that shows how complex a model is, the other term $C\sum_{i=1}^{m} l(y_i, f(\mathbf{x}_i))$ refers to **training errors** (*loss function*).

Hinge loss

In order to use the *regularization theory* we have to specify the *loss function*.

In *soft margin SVM* what we minimize is $\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m}\xi_i$, implying that $\xi_i = l(y_i, f(\mathbf{x}_i))$ and the constraint is $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1 - \xi_i$ rewritable as $\xi_i \geq 1 - y_i(\mathbf{w}^T\mathbf{x}_i + w_0)$ remembering that $\xi_i \geq 0$:

$$l(y_i, f(\mathbf{x}_i)) = |1 - y_i(\mathbf{w}^T\mathbf{x}_i + w_0)|_+$$
$$|z|_+ \text{ means it is } = z \text{ if } z > 0, 0 \text{ otherwise}$$

> $\xi_i = l(y_i, f(\mathbf{x}_i))$
>
> ξ_i is the hinge loss
>
> $yf(\mathbf{x})$ is the confidence

$\rightarrow$ the loss is 0 if the confidence in the right prediction is at least 1,
if it's less than 1 the loss is $1 - yf(\mathbf{x})$.

Lagrangian and Dual in Soft margin SVM

$$L = C\sum_{i=1}^{m}\xi_i + \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m}\alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + w_0) - 1 + \xi_i) - \sum_{i=1}^{m}\beta_i\xi_i$$

β_i is the **Lagrange multiplier** for the constraint $\xi_i \geq 0$

$$\nabla_w L = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i \rightarrow \mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

So we get:

- $\frac{\delta L}{\delta w_0} = 0 \rightarrow \sum_{\alpha_i, y_i}^{m} = 0$
- $\frac{\delta L}{\delta \mathbf{w}} = 0 \rightarrow \mathbf{w} + \sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i$
- $\frac{\delta L}{\delta \xi_i} = 0 \rightarrow C - \alpha_i - \beta_i = 0$

In the end, substituting in the *Lagrangian* we get: $L(\alpha) = \sum_{i-1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j+1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j$

So the **dual formulation** is

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & \sum_{i-1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned}$$

Different from hard margin SVM because α is upper-bounded by C

In the Lagrangian, when we get to the saddle point, the result $\forall i$ is

$$\begin{aligned} \alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + w_0) - 1 + \xi_i) &= 0 \\ \beta_i\xi_i &= 0 \\ C - \alpha_i - \beta_i &= 0 \end{aligned}$$

thus, *support vectors* (where $\alpha_i > 0$) are examples for which $y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \leq 1$.

Then, let's suppose $\alpha_i > 0$, if $\alpha_i < C$, $\beta_i > 0$ and then of course $\xi_i = 0$: these support vectors are called **unbound SV**, because they stay in the confidence $= 1$ hyperplane ($y_i f(\mathbf{x}_i) = y_i(\mathbf{w}^T\mathbf{x}_i + w_0) = 1$)

If $\alpha_i = C$, they are called **bound SV**, and then ξ_i can be greater than zero: in such case the SV are *margin errors*.

Margin errors can be also *training errors* if they are in the wrong side of the hyperplane.

Large-scale SVM

Training of SVM is a quadratic optimization problem; if the dataset is large, to train more quickly there is the **stochastic gradient descent**

We take back the objective of SVM $\frac{||\mathbf{w}||^2}{2} + C\sum_{i=1}^{m}\xi_i = \frac{||\mathbf{w}||^2}{2} + C\sum_{i=1}^{m}|1 - y_i(\mathbf{w}^T\mathbf{x}_i + w_0)|_+$ and changing $\frac{m}{C} = \lambda$ we have the *stochastic gradient descent*:

$$\min_{\mathbf{w}\in X}\frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{m}\sum_{i=1}^{m}|1 - y_i\langle\mathbf{w},\mathbf{x}_i\rangle|_+$$

Stochastic means we compute the gradient on a single example at a time:

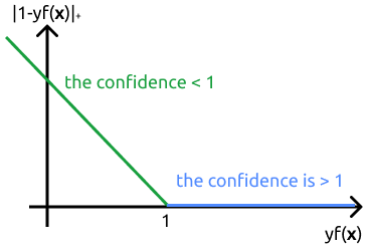
$$(\mathbf{x}_i, y_i) : E(\mathbf{w},(\mathbf{x}_i,y_i)) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + |1 - y_i\langle\mathbf{w},\mathbf{x}_i\rangle|_+$$

Computing subgradient: when you don't have gradient in a point (non derivability point), we can still find some gradients

The **subgradient** of a function f at a point $\mathbf{x}_0$ is any vector $\mathbf{v}$ such that for any $\mathbf{x}$ that this holds: $f(\mathbf{x}) - f(\mathbf{x}_0) \geq \mathbf{v}^T(\mathbf{x} - \mathbf{x}_0)$, it means you can use any of this vector as gradient in points where the derivatives doesn't exists.

Subgradient on such example = $\nabla_{\mathbf{x}}E(\mathbf{w},(\mathbf{x}_i,y_i)) = \lambda\mathbf{w} - \mathbb{1}[y_1\langle\mathbf{w},\mathbf{x}_i\rangle < 1]y_i\mathbf{x}_i$

> indicator function: $\mathbb{1}[y_1\langle\mathbf{w},\mathbf{x}_i\rangle < 1] = \begin{cases}1 & \text{if } y_i\langle\mathbf{w},\mathbf{x}_i\rangle < 1\\ 0 & \text{otherwise}\end{cases}$



The algorithm to do the large scale learning is called *Pegasus*:

$$\mathbf{w}_1 = 0$$
$$\text{for } t = 1 \text{ to } T:$$
$$\begin{cases}1.\ \text{randomly choose } (\mathbf{x}_{i_t}, y_{i_t}) \text{ from } D\\ 2.\ \text{set } \eta_t = \frac{1}{\lambda t}\\ 3.\ \text{update } \mathbf{w} \text{ with } \mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t\nabla_{\mathbf{w}}E(\mathbf{w},(\mathbf{x}_{i_t},y_{i_t}))\end{cases}$$
$$\text{return } \mathbf{w}_{T+1}$$

The *learning rate* is not static, it's an **adaptive learning rate** that decreases with t: The choice of the learning rate allows to bound the runtime for an ϵ-accurate solution to $\mathrm{O}(d/\lambda\epsilon)$ with d maximum number of non-zero features in an example.

Extra

Dual version

$$\mathbf{w}_{t+1} = \frac{1}{\lambda t}\mathbb{1}[y_{i_t}\langle\mathbf{w}_t,\mathbf{x}_{i_t}\rangle < 1]y_{i_t}\mathbf{x}_{i_t}$$

We can represent $\mathbf{w}_{t+1}$ implicitly by storing in vector α_{t+1} the number of times each example was selected and had an on-zero loss, i.e. $\alpha_{t+1}[j] = |\{t' \leq t : i_{t'} = j \wedge y_j\langle\mathbf{w}_{t'},\mathbf{x}_j\rangle < 1\}|$

There is a version of Pegasus for the dual, useful combined with kernels.

Non-linear Support Vector Machines

So far, hard margin assumes that training examples are linearly separable and separates them with the larger margin.

Soft margin SVM allows for exceptions in hard margin.

Non-linear separable problems need a higher expressive power and we don't want to loose the advantages of linear separators (such as large margin and theoretical guarantees): the solution is to *map* input examples in a *higher dimensional feature space* and perform linear classification in this higher dimensional space.

ϕ is a **mapping function** mapping each example to a higher dimensional space $\phi : X \to H$ where examples $\mathbf{x}$ are replaced with $\phi(\mathbf{x})$; this should *increase the expressive power* of the representation (creating new features as combinations of input features): examples should be linearly separable in the new mapped space.

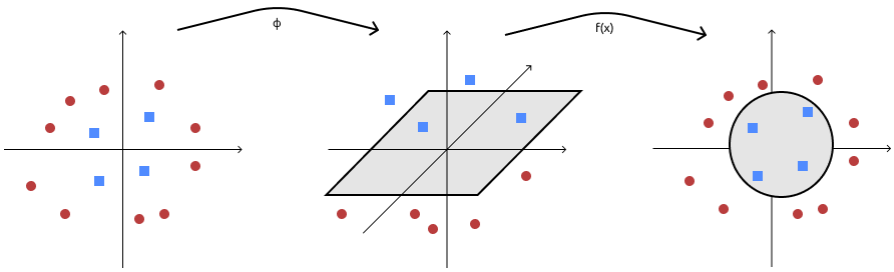
> in some cases, the H Hilbert Space can be infinitely dimensional, we will see in future topics

Polynomial Mapping

The function maps input vector to their **polynomial**, either:

- of a certain degree d (homogeneous mapping) $\to \phi \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x^2 \\ xy \\ y^2 \end{pmatrix}$
- up to a certain degree (inhomogeneous mapping). $\to \phi \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x^2 \\ x \\ xy \\ y \\ y^2 \end{pmatrix}$ (there is even the degree 1 other than the degree 2)

SVM algorithm is applied simply replacing $\mathbf{x}$ with $\phi(\mathbf{x})$ in $f(\mathbf{x}) \to f(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) + w_0$
A linear separation in feature space correspond to a non-linear separation in input space:



Support Vector Regression

Non linear SVM used for **regression**: we want to combine and trade of the *fitting of the examples* and smoothness of the function (*complexity*).

> mean squared error -> $\frac{|f(\mathbf{x}) - y|^2}{m}$
>
> in SVM you don't pay any cost ONLY if the value is exactly y

In regression we want a loss function **more tolerant**, that tolerates small ϵ deviation from the true value with no penalty:
the ϵ-insensitive loss:

$$l(f(\mathbf{x}, y)) = |y - f(\mathbf{x})|_\epsilon = \begin{cases} 0 & \text{if } |y - f(\mathbf{x})| \leq \epsilon \\ |y - f(\mathbf{x})| - \epsilon & \text{otherwise} \end{cases}$$

if the difference between y and $f(\mathbf{x})$ is within ϵ ($-\epsilon \leq y - f(x) \leq \epsilon$) you don't pay anything; the tolerance is needed because data from measurements or people are almost impossibly perfect.

- Defines an ϵ-tube of insensitiveness around true values.
- ϵ is an *hyperparameter*, so you have to decide it before the learning/prediction.

This allows to trade off function complexity with data fitting (playing on ϵ value).

The **optimization problem is**:

$$\begin{gathered} \max_{\mathbf{w} \in X, w_0 \in \mathbb{R}, \xi, \xi^* \in \mathbb{R}^m} \quad \frac{\|\mathbf{w}\|^2}{2} + C \sum_{i=1}^{m} (\xi_i + \xi_i^*) \\ \text{subject to} \quad \mathbf{w}^T \phi(\mathbf{x}_i) + w_0 - y_i \leq \epsilon + \xi_i \\ y_i - (\mathbf{w}^T \phi(\mathbf{x}_i) + w_0) \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{gathered}$$

As we can see, there are two constraints for each example for the upper and lower sides of the ϵ-tube.

Slack variables ξ_i, ξ_i^* penalize predictions out of the ϵ-intensive tube.

Solving with Lagrangian Multiplier (Dual formulation)

Vanishing the derivatives (gradient) with relation to the primal variables we obtain:

$$\mathbf{w} = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*) \phi(\mathbf{x}_i)$$

$$\frac{\delta L}{\delta w_0} = 0 \rightarrow \sum_i (\alpha_i^* - \alpha_i) = 1$$

$$\frac{\delta L}{\delta \xi_i} = C - \alpha_i - \beta_i = 0$$

$$\frac{\delta L}{\delta \xi_i^*} = C - \alpha_i^* - \beta_i^* = 0$$

And substituting in the lagrangian we get (after derivatives and semplifications):

$$\begin{gathered} \max_{\alpha \in \mathbb{R}^m} \quad -\frac{1}{2} \sum_{i,j=1}^{m} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) - \epsilon \sum_{i=1}^{m} (\alpha_i^* + \alpha_i) + \sum_{i=1}^{m} y_i (\alpha_i^* - \alpha_i) \\ \text{subject to} \quad \sum_{i=1}^{m} (\alpha_i - \alpha_i^*) = 0 \\ \alpha_i, \alpha_i^* \in [0, C] \quad \forall i \in [1, m] \end{gathered}$$

Regression function

replacing what we found in the "classification" function, we get:

$$f(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) + w_0 = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)\phi(\mathbf{x}_i)^T\phi(\mathbf{x}) + w_0$$

> using $\mathbf{x}$ instead of $\phi(\mathbf{x})$ it would be linear regression

Karush-Khun-Tucker conditions

In terms of support vectors, we have (iconstraints):

- at the saddle point it holds that $\forall i$:

$$\alpha_i(\epsilon + \xi_i + y_i - \mathbf{w}^T\phi(\mathbf{x}_i) - w_0) = 0$$
$$\alpha_i^*(\epsilon + \xi_i^* - y_i + \mathbf{w}^T\phi(\mathbf{x}_i) + w_0) = 0$$
$$\beta_i\xi_i = 0$$
$$\beta_i^*\xi_i^* = 0$$

- combining with $C - \alpha_i - \beta_i = 0, \quad \alpha_i \geq 0, \beta_i \geq 0$ and $C - \alpha_i^* - \beta_i^* = 0, \quad \alpha_i^* \geq 0, \beta_i^* \geq 0$
- we get $\alpha_i \in [0, C] \quad \alpha_i^* \in [0, C]$ and $\alpha_i = C$ if $\xi_i > 0 \quad \alpha_i^* = C$ if $\xi_i^* > 0$

As result we have all examples inside the ϵ-tube ($\alpha_i, \alpha_i^* = 0$) that don't contribute to the solution (they are **not support vectors**)

Examples for which either $0 < \alpha_i < C$ or $0 < \alpha_i^* < C$ (they stay on the border) are called **unbound support vectors**

The remaining examples that stay out of the ϵ-insensitive region (out of the tube), in such case they are **bound support vectors** and their $\alpha_i = C$ or $\alpha_i^* = C$

> in the regression you have to stay *inside* the tube, in classification *outside* the margin
>
> ma ora che ci penso, cosa cambia tra α_i e α_i^*? Si riferiscono a dei vettori particolari?

Kernel Machines

For learning non-linear models we can apply feature mapping (you have to know *which* mapping to apply). Even if polynomial mapping is useful, in general it could be *expensive* to explicitly compute the mapping and deal with a high dimension feature space.

If we look at dual formulation of SVM problem, the feature mapping only appears in dot products.

The **kernel trick** replace the dot product with an equivalent kernel function over the inputs, that produces the output of the dot product but in feature space, without mapping (explicitly) $\mathbf{x}$ and $\mathbf{x}'$ to it.

The *dual problem* of SVM classification becomes:

$$\max_{\alpha \in \mathbb{R}^m} \quad \sum_{i-1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \overbrace{\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}^{k(\mathbf{x}_i, \mathbf{x}_j)}$$
$$\text{subject to} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m$$
$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

and the *decision function* becomes : $f(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i y_i \overbrace{\phi(\mathbf{x}_i)^t \phi(\mathbf{x})}^{k(\mathbf{x}_i, \mathbf{x})}$

Example in polynomial mapping

- **Homogeneous** polynomial kernel: the same result can be achieved by taking as kernel $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}')^d$ it is the dot product in input space not feature space and the result is a scalar raised to d
- **Inhomogeneous** polynomial kernel: $k(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T \mathbf{x}')^d$

> Using as example
> $k(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}) = (x_1 x'_1 + x_2 x'_2)^2$ and
> $k(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}) = (1 + x_1 x'_1 + x_2 x'_2)$

Kernel validity

A kernel, if valid, always corresponds to a dot product in some feature space.

A kernel is valid if it's defined as a *similarity function* defined as **cartesian product** of input space $k : X \times X \to \mathbb{R}$. It corresponds to a dot product in a certain feature space
$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$

You can compute kernels even with objects that are not vectors, like sequences $\rightarrow$ the kernel generalizes the notion of dot product to arbitrary input space

Condition for validity

The **Gram matrix** is a *symmetric matrix* of the kernels between pairs of examples:
$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) \quad \forall i, j$

Positive definite matrix: a symmetric matrix is positive definite if for any possible vector c

$$\sum_{i,j=1}^{m} c_i c_j K_{ij} \geq 0, \quad \forall \mathbf{c} \in \mathbb{R}^m$$

If equality only holds for $c = 0$, the matrix is **strictly positive definite**

Alternative definitions:

- all eigenvalues of K are non-negative
- there exists a matrix B such that $K = B^T B$

Validity of Kernel

Conditions for a kernel to be valid:

Positive definite kernel

- A positive definite kernel is a function $k : X \times X \to \mathbb{R}$ giving rise to a positive definite *Gram matrix* for any m and $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$
- positive definiteness is **necessary** and **sufficient** condition for a kernel to correspond to a dot product of some feature map ϕ

To verify if the kernel is valid, either we have to:

- show how is the ϕ
- make the *feature map explicit*
- using *kernel combination properties* (using already valid kernels combined)

In the dual problem for SVM regression, $\phi(\mathbf{x})$ appears only in the dot product (and also in the decision function $f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + w_0 = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*) \overbrace{\phi(\mathbf{x}_i)^T \phi(\mathbf{x})}^{k(\mathbf{x}_i, \mathbf{x})} + w_0$):

$$\max_{\alpha \in \mathbb{R}^m} \quad -\frac{1}{2} \sum_{i,j=1}^{m} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) \overbrace{\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}^{k(\mathbf{x}_i, \mathbf{x}_j)} - \epsilon \sum_{i=1}^{m} (\alpha_i^* + \alpha_i) + \sum_{i=1}^{m} y_i (\alpha_i^* - \alpha_i)$$

$$\text{subject to} \quad \sum_{i=1}^{m} (\alpha_i - \alpha_i^*) = 0$$

$$\alpha_i, \alpha_i^* \in [0, C] \quad \forall i \in [1, m]$$

Kernelize the Perceptron

- *stochastic perceptron* (already seen) $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$:
  - initialize $\mathbf{w} = 0$
  - iterate until all examples are classified correctly $\rightarrow$ for each incorrectly classified training example $(\mathbf{x}_i, y_i) : \mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
- **kernel perceptron** $f(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i k(\mathbf{x}_i, \mathbf{x})$:
  - initialize $\alpha_i = 0 \quad \forall i$
  - iterate until all examples are classified correctly $\rightarrow$ for each incorrectly classified training example $(\mathbf{x}_i, y_i) : \alpha_i \leftarrow \alpha_i + \eta y_i$

$\Rightarrow$ in kernel machines you always have that the decision function is the *sum over the training sample of coefficient times* $k(\mathbf{x}_i, \mathbf{x})$. The coefficient changes depending on the problem.

Basic kernels

- **linear kernel** $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}'$: it computes only the dot product: it's useful when combining kernel because when you don't have a kernel, in reality you have the linear kernel;
- **polynomial kernel** $k_{d,c}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}' + c)^d$: homogeneous if $c = 0$, inhomogeneous otherwise
- **gaussian kernel** $k_\sigma(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\mathbf{x}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{x}' + \mathbf{x}'^T\mathbf{x}'}{2\sigma^2}\right)$
  - depends on σ, the **width** parameter
  - the smaller the width, the more predictions on a point only depends on its nearest neighbours
  - it's a **universal kernel**: it can uniformly approximate any arbitrary continuous target function (but you first have to find the correct value of σ)

> Gaussian has infinite dimensional feature space

The **choice** of a Kernel is made **before** training.

Kernel on structured data

We don't need to think input as vectors anymore: now the kernel can be seen as a way to *generalize dot products to arbitrary domains*. It's possible to design kernels over structured objects such as sequences, graphs and trees.

The idea is to create a pairwise function measuring the **similarity between two objects**. This measure has to satisfy the *valid kernel conditions*.

Examples of Kernel over structure

Every kernel over structure is built as combination of kernels over pieces.

The simplest **kernel over pieces** (of a structure) is the **delta kernel** (or *match* kernel)
$k_\delta(x, x') = \delta(x, x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$ where x does <u>not</u> need to be a vector

Kernel on sequeces

Spectrum kernel: (looking at frequencies of subsequences of n symbols) and the result is the number of time each subsequence appears in the starting sequence. The feature space is the space of all possible k-subsequences.

Kernel Combinations

Building the kernel combining pieces, provided the basic kernels are valid and the operations are valid, the result is valid.

- **sum**: concatenating the features of the two kernels (that are valid); can be defined on different spaces (k_1 and k_2 could look at different things):

$$\begin{aligned}(k_1 + k_2)(x, x') &= k_1(x, x') + k_2(x, x') \\ &= \phi_1(x)^T\phi_1(x') + \phi_2(x)^T\phi_2(x') \\ &= (\phi_1(x)\phi_2(x)) \begin{pmatrix}\phi_1(x') \\ \phi_2(x')\end{pmatrix}\end{aligned}$$

- **product**: the resulting feature space is the cartesian product between the feature space of the first kernel and the feature space of the second kernel

$$
\begin{aligned}
(k_{\times} k_1)(x, x') &= k_1(x, x') k_2(x, x') \\
&= \sum_{i=1}^{n} \phi_{1i}(x)\phi_{1i}(x') \sum_{j=1}^{m} \phi_{2j}(x)\phi_{2j}(x') \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} (\phi_{1i}(x)\phi_{2j}(x))(\phi_{1i}(x')\phi_{2j}(xi)) \\
&= \sum_{k=1}^{nm} \phi_{12k}(x)\phi_{12k}(x') = \phi_{12}(x)^T \phi_{12}(x')
\end{aligned}
$$

  - where $\phi_{12}(x) = \phi_1 \times \phi_2$ is the Cartesian product
  - the product can be between kernels in different spaces (**tensor** product)
- **linear combination**: you can always multiply by a positive scalar (if it's negative the kernel becomes invalid). If I don't know which kernel to use I just take a bunch of them, combine them linearly and learn the parameters (in addition to learn the αs of the dual) $\rightarrow$ this is called **kernel learning**
  - a kernel can be rescaled by an arbitrary positive constant $k_\beta(x, x') = \beta k(x, x')$
  - we can define, for example, linear combinations of kernels (each scaled by a desired weight): $k_{\text{sum}}(x, x') = \sum_{k=1}^{k} \beta_k k_k(x, x')$
- **normalization**: kernel values can often be influenced by the dimension of objects (a longer string contains more substrings $\rightarrow$ higher kernel value). This effect can be reduced with *normalizing* the kernel.
  Cosine normalization computes the cosine of the dot product in feature space:
  $\hat{k}(x, x') = \dfrac{k(x, x')}{\sqrt{k(x, x) k(x', x')}}$
- **composition**: you can combine also by composition

$$
\begin{aligned}
(k_{d,c} \ \circ \ k)(x, x') &= (k(x, x') + c)^d \\
(k_{\sigma} \ \circ \ k)(x, x') &= \exp\left(-\frac{k(x, x) - 2k(x, x') + k(x', x')}{2\sigma^2}\right)
\end{aligned}
$$

  it corresponds to the composition of the mappings associated with the two kernels.

Kernel on Graphs

Wistfeiler-Lehman graph kernel

It's an efficient **graph kernel** for large graphs. It relies on (approximation of) Weistfeiler-Lehman test of graph isomorphism and it describes a family of graph kernel:

- Let $\{G_0, G_1, \dots, G_h\} = \{(V, E, I_0), (V, E, I_1), \dots, (V, E, I_h)\}$ be a sequence of graphs made from G where I_i is the node labelling the ith WL iteration
- Let $k : G \times G' \rightarrow \mathbb{R}$ be any kernel on graphs
- the Weistfeiler-Lehman graph kernel is defined as: $k_{WL}^h(G, G') = \sum_{i=0}^{h} k(G_i, G_i')$

> Graphs G and H are isomorphic if there is a structure that preserves a one-to-one correspondence between the vertices and edges.

Unsupervised Learning

Setting:

- supervised learning requires the availability of labelled examples (an extremely expensive process)
- sometimes it is not even known how to label examples
- **unsupervised** techniques can be employed to group examples into **clusters** (groups)

It's often the first step when looking at some data.

k-means

Simplest approach for *clustering* is k-means; we decide a priori the number of clusters k. Each cluster i will be represented by its mean μ_i.

Idea: you want examples to be in cluster with the *closest* mean.

Algorithm:

1. **initialize** the cluster: randomly (for example picking k random examples)
2. iterate until there is **no change in the means**
    - for each example take its nearest mean and assign it to such *cluster*
    - a the end of iteration you compute the *new means*

Usually there is some work to do before (like *normalization*).

Define the quality of the clusters

Dissimilarity measures

- (if samples are vectors) Standard Euclidean distance in $\mathbb{R}^d$: $d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{i=1}^{d}(x_i - x_i')^2}$
- Generic Minkowski metric for $p \geq 1$ (where p is an integer): $d(\mathbf{x}, \mathbf{x}') = \left(\sum_{i=1}^{d}|x_i - x_i'|^p\right)^{\frac{1}{p}}$
- Cosine similarity (cosine of the angle between vectors): $s(\mathbf{x}, \mathbf{x}') = \frac{\mathbf{x}^T\mathbf{x}'}{\|\mathbf{x}\|\|\mathbf{x}'\|}$

Metric learning is important: instead of assuming a predefined metric, you *learn* it from data (maybe some feature have different weights)

Sum of squared error criterion

- let n_i be the number of samples in cluster D_i
- let μ_i be the sample mean $\mu_i = \frac{1}{n_i}\sum_{\mathbf{x}\in D_i}\mathbf{x}$

  $\Rightarrow$ the sum-of-squared error is defined as: $E = \sum_{i=1}^{k}\sum_{\mathbf{x}\in D_i}\|\mathbf{x} - \mu_i\|^2$

It measures the squared error incurrent in representing each sample with its cluster mean.

(There are other quality measure)

Clustering Approach: Gaussian Mixture Model

We decide a priori the number of *clusters*. Each cluster is represented using a Gaussian distribution $\rightarrow$ we need to estimate the *mean* and the *variance* for each Gaussian.

Problem: if you have data and you have to fit the Gaussian, you should make Maximum-Likelihood of parameters (mean and variance) but you don't know each example at which gaussian it corresponds; so we adopt the **Expectation-Maximization** approach:

1. Compute *expected cluster* assignment given the current parameter setting
2. *Estimate parameters* given the cluster assignment
3. *Iterate*

Estimating means of k univariate Gaussian

Setting (the latent variable is the cluster assignment):

- A dataset of $x_1, \dots, x_n$ examples is observed
- For each example x_i, cluster assignment is modelled as $z_{i1}, \dots, z_{ik}$ binary latent (unknown) variables
- $z_{ij} = 1$ if Gaussian j generated x_i, 0 otherwise [*hot encoding*]
- Parameters to be estimated are the $\mu_1, \dots, \mu_k$ Gaussians means
- All Gaussians are assumed to have the same (known) variance σ^2

> latent variable = not directly observed but rather inferred (through a mathematical model) from other variables
>
> hot encoding: a vector of k binary variables, $z_{ij} = 1$ if the example x_i came from gaussian j , 0 otherwise (it represent the cluster assigned)

Algorithm:

1. initialize the parameters (as hypothesis) $h = \langle \mu_1, \dots, \mu_k \rangle$
2. Iterate until the difference in *Maximum Likelihood* is below a certain threshold:
   - **E-step**: calculate expected value $E[z_{ij}]$ of each latent variable assuming current hypothesis $h = \langle \mu_1, \dots, \mu_k \rangle$ holds
   - **M-step**: calculate a new ML hypothesis $h' = \langle \mu'_1, \dots, \mu'_k \rangle$ assuming values of latent variables are their expected values just computed. Replace $h \leftarrow h'$ (at the end of this step we have a new hypothesis that replaces the current, and repeat)

Computing the algorithm:

- **E-step**: The expected value of z_{ij} is the probability that x_i is generated by Gaussian j assuming hypothesis $h = \langle \mu_1, \dots, \mu_k \rangle$ holds:
  $$E[z_{ij}] = \frac{p(x_i|\mu_j)}{\sum_{l=1}^{k} p(x_i|\mu_l)} = \frac{\exp\left(-\frac{1}{2\sigma^2}[x_i - \mu_j]^2\right)}{\sum_{l=1}^{k} \exp\left(-\frac{1}{2\sigma^2}[x_i - \mu_l]^2\right)}$$
  (the coefficient of the gaussian is absent because is cancelled out with the denominator because it's constant)
- **M-step**: The maximum-likelihood mean μ_j is the weighted sample mean, each instance being weighted by its probability of being generated by Gaussian j: $\mu'_j = \frac{\sum_{l=1}^{n} E[z_{ij}] x_i}{\sum_{l=1}^{n} E[z_{ij}]}$

EM is a general strategy for dealing with *optimization of maximization* of parameters:

- We are given a dataset made of an observed part X and an unobserved part Z
- We wish to estimate the hypothesis maximizing the expected log-likelihood for the data, with expectation taken over unobserved data: $h^* = \text{argmax}_h E_Z[\ln p(X, Z|h)]$

> EM is guaranteed to converge but it could find a local optimum of the likelihood

The unobserved data Z should be treated as random variables that depends on X and h

How the generic algorithm works and how it is instantiated in the gaussian mixture model problem:

- initialize hypothesis h
- iterate until convergence:
  - **E-step**: computes the expected likelihood of an hypothesis h' for the full data, where the unobserved data distribution is modelled according to the current hypothesis h and the observed data: $Q(h', h) = E_Z[\ln p(X, Z|h')|h, X]$

    > expected likelihood of h' for the full data, where observed data is the actual observed data, and the unobserved is modelled according to current hypothesis h
    >
    > log lof the likelihood of the data, both observed and unobserved, given the new hypothesis; this likelihood is computed in expectation over Z, and the expectation is computed according to the current versione of hypothesis and the observed data (this configuration is similar to something with bayesian network)

  - **M-step**: replace the current hypothesis with the new one maximizing $Q(h', h)$: $h \leftarrow \text{argmax}_{h'} Q(h', h)$

This is what happens in the case of Gaussian mixture model (derivation):

- the likelihood of an example is:
  $$p(x_i, z_{i1}, \dots, z_{ik}|h') = \frac{1}{\sqrt{2\pi\sigma}} \exp\left[-\sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2}\right]$$

  > if z is one hot, only the gaussian with 1 will be recovered.
  >
  > The sum is weighted by z

- the dataset log-likelihood is: $\ln p(X, Z|h) = \sum_{i=1}^{n} \left(\ln \frac{1}{\sqrt{2\pi\sigma}} - \sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2}\right)$

  > in log, the product becomes a sum

- **E-step**:
  - Expectation over Z of $\ln p(X, Z|h')$ is:
    $$E_Z[\ln p(X, Z|h')] = E_Z\left[\sum_{i=1}^{n} \left(\ln \frac{1}{\sqrt{2\pi\sigma}} - \sum_{j=1}^{k} z_{ij} \frac{(x_i - \mu'_j)^2}{2\sigma^2}\right)\right] =$$
    $$\sum_{i=1}^{n} \left(\ln \frac{1}{\sqrt{2\pi\sigma}} - \sum_{j=1}^{k} E[z_{ij}] \frac{(x_i - \mu'_j)^2}{2\sigma^2}\right)$$

    > expectation is a linear operator $\rightarrow$ the expectation of two variables is the sum of the two expectations

> everything that's not z is a constant in terms of expectation because this is the expectation of z

- Using the current version of the hypothesis, expectation is computed as:

$$E[z_{ij}] = \frac{p(x_i|\mu_j)}{\sum_{l=1}^{k} p(x_i|\mu_l)} = \frac{\exp -\frac{1}{2\sigma^2}(x_i - \mu_j)^2}{\sum_{l=1}^{k} \exp -\frac{1}{2\sigma^2}(x_i - \mu_l)^2}$$

> the expectation is the density for gaussian j normalized over all possible gaussians (the probability that x_i comes from that gaussian)

- **M-step**:
  - the maximization over h':

$$\text{argmax}_{h'} Q(h', h) = \text{argmax}_{h'} \sum_{i=1}^{n} \left(\ln \frac{1}{\sqrt{2\pi\sigma}} - \sum_{j=1}^{k} E[z_{ij}] \frac{(x_i - \mu'_j)^2}{2\sigma^2} \right) =$$

$$\text{argmin}_{h'} \sum_{i=1}^{n} \sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_i)^2$$

> the first part doesn't contain μ' so it's constant so we can get rid of it. Removing the minus makes it argmin

$$\frac{\delta}{\delta \mu_j} = -2 \sum_{i=1}^{n} E[z_{ij}](x_i - \mu'_j) = 0$$

  - so I zero the derivative and find μ'

$$\Rightarrow \mu'_j = \frac{\sum_{i=1}^{n} E[z_{ij}] x_i}{\sum_{i=1}^{n} E[z_{ij}]}$$

How to choose the number of clusters

Until now we took the number of clusters as given and known in advance, but usually it's not obvious.

Maybe our knowledge gives us an intuition on the number of clusters, but we can try to find out the number of clusters.

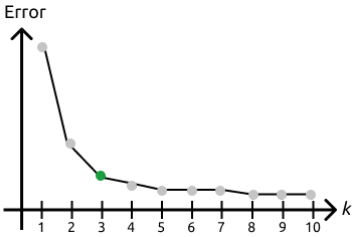
Elbow method

Very simple method. If we use the *mean squared error*, increasing the number of clusters always decreases the error. We need to **trade-off quality of clusters with quantity** of clusters, so we stop increasing number of clusters when the advantage is limited.

Approach:

1. **Run** clustering algorithm for increasing number of clusters
2. **Plot** the clustering evaluation metric, for different k and we connect the dots
3. Try to see where the curve that is going down has an **elbow**, so when the error starts to be less important and we choose that number of clusters

It's a bit ambiguous method because the choice is made by eye, or also computing the angle, but there is no formal definition on when to stop.



Average Silhouette

The idea: Increasing the numbers of clusters makes each cluster more homogeneous internally, but also different clusters become more similar. The silhouette method computes this **dissimilarity** between clusters, to *trade-off intra-cluster similarity and inter-cluster dissimilarity*.

How it works (for an example i):

1. Compute the average dissimilarity between i and examples of *its* cluster C:
   $a_i = d(i, C) = \frac{1}{|C|} \sum_{j \in C} d(i, j)$

   > it's the average distance between i and the examples of its cluster (j is every other example), it's made for each i - intra cluster similarity

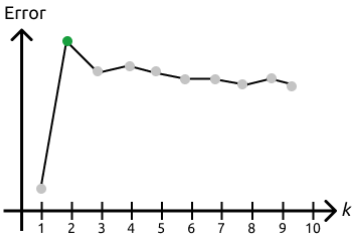
2. Compute the average dissimilarity between i and examples of *each* cluster $C' \neq C$, and take the minimum: $b_i = \min_{C' \neq C} d(i, C')$

   > inner cluster dissimilarity, min = with the closest cluster

3. The silhouette coefficient is: $s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$

   > the difference between the two values, normalized by the maximum

Then, I run the clustering algorithm up to a certain point, I compute the silhouette for each clustering, I average it over all example and I plot it. I choose k where the average silhouette coefficient is maximal



Extra

In **Gaussian Mixture Model** there is also another solution based on Bayesian non parametric.

GMM assumes you already know k, in Bayesian terms we can combine GMM with the prior on the number of clusters. It computes *for every possible values* of k and allows you to model options in which GMM can have different number of gaussians and depending on the priori process looks like, it *favours number of gaussians that well fits your data* combined with a *small number of clusters that it's embedded with priori* - It's Bayesian non-parametric, instead of having parameters, it adds a prior process that automatically adjust the value of parameters

→ **Dirichlet process mixture model**

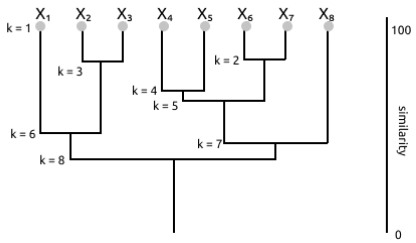
Hierarchical clustering

Up to now we took our data and divided them in k clusters (flat). In many situation, data groups have a hierarchy.

A **hierarchy** of clusters can be built in greedy ways:

- **Top-down**:
  1. start with a single cluster with all examples
  2. recursively split clusters into subclusters, accordingly to a certain measure and repeat
- **Bottom up**:
  1. start with n clusters of individual examples (singletons)
  2. recursively aggregate pairs of clusters (the closest groups everytime)

Dendrogram (an example of bottom up):



Agglomerative hierarchical clustering (bottom-up)

Algorithm:

1. Initialize:
   - final number of cluster k
   - initial number $\hat{k} = n$
   - initial clusters $D_i = \{x_i\},\ i \in 1, \ldots, n$
2. while $\hat{k} > k$:
   - find the pair of closes clusters D_i, D_j (based on some similarity measure)
   - merge those clusters into a single cluster
   - $\hat{k} = \hat{k} - 1$

You can stop whenever the problem requires.

Similarity measures

Between clusters/sets

- **Nearest-neighbour**: *minimal distance* between the elements in the two sets
  $d_{\min}(D_i, D_j) = \min_{\mathbf{x} \in D_i, \mathbf{x}' \in D_j} \|\mathbf{x} - \mathbf{x}'\|$
- **Farthest-neighbour**: *maximal distance* between the elements in the two sets
  $d_{\max}(D_i, D_j) = \max_{\mathbf{x} \in D_i, \mathbf{x}' \in D_j} \|\mathbf{x} - \mathbf{x}'\|$
- **Average distance**: *average the distances* between elements in the two sets
  $d_{\text{avg}} = \frac{1}{n_i n_j} \sum_{\mathbf{x} \in D_i} \sum_{\mathbf{x}' \in D_j} \|\mathbf{x} - \mathbf{x}'\|$
- **Distance between means**: just the *distance between the two means*
  $d_{\text{mean}}(D_i, D_j) = \|\mu_i - \mu_j\|$

$d_{\min}$ and $d_{\max}$ are more sensitive to outliers

d_{avg} distance can be expensive

Stepwise optimal hierarchical clustering

It uses some external quality metric.

Algorithm: it's the same as before (agglomerative hierarchical clustering) but using the external metric for choosing which cluster to merge.

The problem is, that even this approach is greedy.

Reinforcement Learning

It's a learning setting, where the learner is an **Agent** that can perform a set of **actions** A depending on its state in a set of **states** S and the environment. In performing action a in state s, the learner receive an immediate **reward** $r(s, a)$.

In some states, some actions could be *not possible/valid*.

The task is to **learn a policy** allowing the agent to choose for each state s the action a **maximizing the overall reward**, including future moves.

To deal with this delayed reward problem, the agent has to trade-off *exploitation* and *exploration*:

- **exploitation** is action it knows give some rewards
- **exploration** experience *alternative* that could end in bigger reward

This involves also a task called **credit assignment**, to understand which move was responsible for a positive or negative reward.

> like playing a game and see the win or lose to understand if it was a bad move and in case don't repeat it

Sequential Decision Making

Setting:

- An agent needs to take a **sequence of decisions**
- The agent should **maximize** some **utility function**
- There is **uncertainty** in the result of a **decision**: if you take an action, it's not deterministic that such action brings the agent to a certain state

Formalization with Markov Decision Process

We have:

- a set of state S in which the agent can be at each instant
- A set of terminal states $S_G \subset S$ (even empty)
- A set of actions A
- a transition model providing the probability of going to a state s' with action a from state s: $P(s'|s, a) \quad s, s' \in S,\ a \in A$
- a reward $R(s, a, s')$ for making action a in state s which led the agent to state s'

Defining Utilities

We need to **define utilities** in order **to find the policy** for the agent, since it has to *maximize the reward*, and we need to formalize what it is the overall reward (so the utilities over time). We need:

- An **environment history** is a *sequence* of states, since we are not only interested in immediate rewards but also in delayed rewards
- **Utilities** are defined over environment histories
- We assume an **infinite horizon** (a priori *no constraint on the number of time steps*)
- We assume **stationary preferences** (if one history is preferred to another at time t, the same should hold at time t' provided they start from the same state)

Utilities over time

Utility is defined over history of states, two simple way to define utilities:

- **Additive rewards**: sum up each reward for each state in the sequence:
  $U([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$

  > to make easier, $R(s_n)$ gives the reward of $R(s, a, s_n)$

- **Discounted rewards** $\gamma \in [0, 1]$ is a decaying factor:
  $U([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$

The Policy

A policy π is a full specification of *what action* to take *at each state*.

The utility in a non deterministic setting: expected utility of a policy; is the utility of an environment history, taken in expectation over all possible histories generated with that policy

An **optimal policy** π^* is a policy maximizing expected utility (since starting from the same state doesn't always produce the same state due to the non-deterministic factor)

For infinite horizons, optimal policies are stationary: they only depend on the current state (at a certain state it decide the action based only on the current state)

Discussion on optimal policy

- If moving is very *expensive*, optimal policy is to *reach any terminal* state asap
- If moving is very *cheap*, optimal policy is *avoiding the bad terminal* state at all costs
- If moving gives *positive reward*, optimal policy is to *stay away from terminal* states (resulting in never reaching an "exit", so $\rightarrow$ discounted rewards)

Optimal Policy: Utilities

Utility of a state

Is the expected reward of moving from that state on, and in order to move you need a policy:

$$U^{\pi}(s) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) | S_0 = s\right]$$

> Utility of a state s given a certain policy π is the Expected reward E_{π}

The **true utility** of a state is its utility using the optimal policy: $U(s) = U^{\pi^*}(s)$

Given the *true utility*, an optimal policy is: $\pi^*(s) = \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|s, a) U(s')$

> taking the action that probabilistically brings me to the best possible state, so I sum up over all possible destination states the probability of reaching this destination state, multiplying by the utility of the state
>
> in order to have an optimal policy you also need the transition model

Computing an Optimal Policy

Formalize the relationship between the utility and policy using the Bellman equation:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} p(s'|s,a)U(s')$$

You have one equation for each state, but we don't know the values of the utilities. The solution are the utilities of the states and are unique, but directly solving the set of equations is hard (non-linearities because of the max).

Value iteration

An iterative approach to solve the Bellman equation:

1. initialize $U_o(s)$ to zero for all s
2. repeat:
    1. do Bellman update for each state s: $U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s' \in S} p(s'|s,a)U_i(s')$
    2. $i \leftarrow i + 1$
3. until *max utility difference below a threshold*
4. return U

U_0 is the utility at the zero-th iteration

In the bellman update you use the utility computed before as it was the true utility

You iterate until utilities don't change more than a threshold

Policy iteration

An alternative to value iteration:

1. initialize π_0 randomly
2. repeat:
    1. policy evaluation, solve set of linear equations:
    $U_i(s) = R(s) + \gamma \sum_{s' \in S} p(s'|s, \pi_i(s))U_i(s') \quad \forall s \in S$ where $\pi_i(s)$ is the action that policy π_i prescribes for state s
    2. policy improvement: $\pi_{i+1}(s) \leftarrow \text{argmax}_{a \in A} \sum_{s' \in S} p(s'|s,a)U_i(s') \quad \forall s \in S$
    3. $i \leftarrow i + 1$
3. until *no policy improvement*
4. return π

> U_i is like an approximation

Dealing with partial knowledge

Until now we assume perfect knowledge of everything (even probability). In most cases some of these information are not known, and those are where reinforcement learning is useful thanks to the exploration part.

Reinforcement learning aims at **learning policies by space exploration**.

policy evaluation: policy is given, environment is learned (passive agent)

policy improvement: both policy and environment are learned (active agent)

Passive because the agent receives the policy and applies it.

Policy evaluation in unknown environment

Adaptive Dynamic Programming (ADP)

Algorithm (loop):

1. initialize s
2. repeat:
    1. Receive reward r, set $R(s) = r$
    2. Choose next action $a \leftarrow \pi(s)$
    3. Take action a, reach step s'
    4. update counts $N_{sa} \leftarrow N_{sa} + 1; \quad N_{s'|sa} \leftarrow N_{s'|sa} + 1$
    5. update transition model $p(s'|s,a) \leftarrow N_{s''|sa}/N_{sa} \quad \forall s'' \in S$
    6. update utility estimate $U \leftarrow \mathrm{PolicyEvaluation}(\pi, U, p, R, \gamma)$
3. until s is terminal

Iterations bring better understanding of the environment in terms of its transitions.

Characteristics

The algorithm performs **maximum likelihood estimation of transition probabilities**

Upon updating the transition model, it calls standard policy evaluation to update the utility estimate (U is initially empty)

Each step is *expensive* as it runs policy evaluation (and the number of steps is huge in order to the agent to learn something)

Temporal-difference (TD)

Approximate solution of ADP to reduce the "expensiveness".

The idea is to avoid running policy evaluation at each iteration, but instead locally update utility:

- If transition from s to s' is observed:
    - If s' was always the successor of s, the utility of s should be $U(s) = R(s) + \gamma U(s')$
    - The temporal-difference update rule updates the utility to get closer to that situation: $U(s) \leftarrow U(s) + \alpha(R(s) + \gamma U(s') - U(s))$

      where α is a learning rate (possibly decreasing over time)

Algorithm (loop):

1. initialize s
2. repeat:
    1. Receive reward r
    2. Choose next action $a \leftarrow \pi(s)$
    3. Take action a, reach step s'
    4. Update local utility estimate: $U(s) \leftarrow U(s) + \alpha(r + \gamma U(s') - U(s))$
3. until s is terminal

If the state is new, you "remember" the reward for reaching it.

Characteristics

- No need for a transition model for utility update
- Each step is much faster than ADP
- Same as ADP on the long run
- Takes longer to converge
- Can be seen as a rough efficient approximation of ADP

Policy learning in unknown environment

Modify those algorithms to also learn the policy instead of only evaluating it.

Policy learning requires combining *learning the environment* and *learning the optimal policy* for the environment: an option is to take ADP and replace the step policy evaluation with policy computation.

There is a problem: The knowledge of the environment is incomplete. A greedy agent usually learns a suboptimal policy (lack of exploration).

Exploration-exploitation trade-off

- Exploitation consists in following promising directions given current knowledge
- Exploration consists in trying novel directions looking for better (unknown) alternatives
- A reasonable trade-off should be used in defining the search scheme:
  - ϵ-greedy strategy: choose a random move with probability ϵ, be greedy otherwise
  - assign higher utility estimates to (relatively) unexplored state-action pairs:

    $$U^+(s) = R(s) + \gamma \max_{a \in A} f\left(\sum_{s' \in S} p(s'|s,a)U^+(s'), N_{sa}\right)$$

    with f increasing over the first argument and decreasing over the second.

> probability $1 - \epsilon$ is greedy, random with ϵ probability
>
> higher utility estimate means give a bonus if that state has not been explored much
>
> N_{sa} number of observed state s and took action a

It's common going with solution in which each iteration runs faster.

TD

To learn the $< \text{utility of the state}, \text{action} >$ pairs (action utility):

- TD policy evaluation can also be adapted to learn an optimal policy
- If TD is used to learn a state utility function, it needs to estimate a transition model to derive a policy
- TD can instead be applied to learn an action utility function $Q(s,a)$ (I don't need the transition model, I incorporate it in Q)
- The optimal policy corresponds to: $\pi^*(s = \text{argmax}_{a \in A} Q(s,a))$

SARSA

The algorithm adapted to *learn an utility-action pair* is SARSA.

Here I'm updating the utility-action pair so I need to look at the next utility-action pair.

SARSA: on-policy TD learning:

1. Initialize s
2. Repeat:
    1. Receive reward r
    2. Choose next action $a \leftarrow \pi^{\epsilon}(s)$
    3. Take action a, reach step s'
    4. Choose action $a' \leftarrow \pi^{\epsilon}(s')$ (this point is added in this variant)
    5. Update local utility estimate $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
3. Until s is terminal

This is an algorithm that explores the space and finds rewards of new states in within and updates the utility estimate of the state-action pair and it use it in order to find the updated policy.

Q-learning

Another option is to not take another action and take the max current utility action pair function:

Q-learning: off-policy TD learning

1. Initialize s
2. Repeat:
    1. Receive reward r
    2. Choose next action $a \leftarrow \pi^{\epsilon}(s)$
    3. Take action a, reach step s'
    4. ~~Choose action $a' \leftarrow \pi^{\epsilon}(s')$ (this point is added in this variant)~~
    5. Update local utility estimate
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \mathbf{max}_{\mathbf{a}' \in \mathbf{A}} Q(s',a') - Q(s,a))$
3. Until s is terminal

SARSA vs Q-learning

- SARSA is **on-policy**: it updates Q using the current policy's action
- Q-learning is **off-policy**: it updates Q using the greedy policy's action (which is NOT the policy it uses to search)
- Off-policy methods are *more flexible*: they can even learn from traces generated with an unknown policy
- On-policy methods tend to *converge faster*, and are easier to use for continuous-state spaces and linear function approximators (see following slides)

> a' is decided according to the current policy. a' is not an action to do in order to reach a space, but it also used to update Q (? non so da dove sia uscita questa riga)

Function Approximation

Main aspect of Deep Reinforcement Learning: until now we thought about a tabula representation of utility functions (states or utility-action pairs).

The space grows while you explore it, and sometimes the space could also be continuous.

So you approximate:

- All techniques seen so far assume a **tabular representation** of *utility functions* (of states or actions)

- Tabular representations do not scale to large state spaces (e.g. Backgammon has an order of 10^{20} states)
- The solution is to rely on **function approximation**: approximate $U(s)$ or $Q(s,a)$ with a parameterized function.
- The function takes a state representation as input (e.g. x, y coordinates for the maze)
- The function allows to generalize to **unseen states** (and it allows to compute the utility of states you never saw)

> It produces the Utility of the state of the utility-action pair

Instead of using a table that for each state gives you a value, you use a function that takes the state represented feature-based and gives the utility multiplied by the weight and the weight are the one to learn.

State utility function approximation: State $s \rightarrow \phi(s)$ [*feature vector*] * θ [*parameter vector*] = **estimated value**

Action utility function approximation:

Q learning: $< state, action > \rightarrow$ Q table $\rightarrow$ Q-value

Deep Q learning: state $\rightarrow$ neural network $\rightarrow \begin{cases} Q - \text{value action } 1 \\ Q - \text{value action } 2 \\ . \\ . \\ Q - \text{value action } N \end{cases}$

How you can do the learning process

If U is a function approximating the table I call it U_θ where θ are the parameters (so we don't have the table).

TD learning: state utility

- TD error $E(s, s') + \frac{1}{2}(R(s) + \gamma U_\theta(s') - U_\theta(s))^2$

- Error gradient with respect to function parameters
  $\nabla_\theta E(s, s') = (R(s) + \gamma U_\theta(s') - U_\theta(s))(-\nabla_\theta U_\theta(s))$

- Stochastic gradient update rule
  $$\theta = \theta - \alpha \nabla_\theta E(s, s')$$
  $$= \theta + \alpha(R(s) + \gamma U_\theta(s') - U_\theta(s))(\nabla_\theta U_\theta(s))$$

1. the $R(s) + \gamma U_\theta(s')$ is what you want your utility to be close to (target), subtracting the current $U_\theta(s)$ (to get the error)

   > squared to penalize more

2. Compute the update step for that utility function with the gradient
3. α is the learning rate

Almost the same on Q-learning: TD learning: action utility (Q-learning):

- TD error $E((s, a), s') + \frac{1}{2}(R(s) + \gamma \max_{a' \in A} Q_\theta(s', a') - Q_\theta(s, a))^2$

- Error gradient with respect to function parameters
  $\nabla_\theta E((s,a),s') = (R(s) + \gamma \max_{a' \in A} Q_\theta(s',a') - Q_\theta(s,a))(-\nabla_\theta Q_\theta(s,a))$

- Stochastic gradient update rule
  $$\theta = \theta - \alpha \nabla_\theta E((s,a),s')$$
  $$= \theta + \alpha \left(R(s) + \gamma \max_{a' \in A} Q_\theta(s',a') - \max_{a' \in A} Q_\theta(s,a) \right) (\nabla_\theta Q_\theta(s,a))$$

Bayesian Networks

BN are probabilistic graphical models.

Probabilistic graphical models are graphical representations of the **qualitative aspects of probability distributions** allowing to:

- visualize the **structure** of a probabilistic model in a simple and intuitive way
- **discover properties** of the model, such as *conditional independencies*, by inspecting the graph
- express **complex computations for inference** and learning in terms of graphical manipulations
- represent multiple probability distributions with the same graph, abstracting from their quantitative aspects (e.g. discrete vs continuous distributions)

The structure is a directed acyclic graph G. Each **node** represent a **random variable**, each **edge** represent a **direct relationship** between variables.

The structure encodes these independence assumptions:
$I_l(G) = \{\forall i \; x_i \perp \text{NonDescendants}_{x_i} | \text{Parents}_{x_i}\}$

$\perp$ means **independent** (from its non descendants given its parents)

Distributions: suppose this variable have a probabilistic relationship between them.

- Let p be a joint distribution over variables X
- $I(p)$ is the set of independence that hold in p (aka in the distribution)
- G is an **independency map** (**I-map**) for p if p satisfies the local independences in G: $I_l(G) \subseteq I(p)$

The probability can be decomposed according to graph, we say that p factorizes according to G if $p(x_1, \dots, x_m) = \prod_{i=1}^{m} p(x_i | Pa_{x_i})$ where Pa_{x_i} are the parents of x_i

And we know that If G is an I-map for p, then p factorizes according to G and vice versa, If p factorizes according to G, then G is an I-map for p.

> Why do we want to factorize? Suppose we want to model a joint distribution over m (binary) variables. We get 2^m configurations. To deal with this huge model we factorize.

Bayesian Networks

A Bayesian Network is a pair (G, p) where p factorizes over G (a DAG) and it is represented as a set of *conditional probability distributions* (cpd) associated with the nodes of G

Factorized Probability: $p(x_1, \dots, x_m) = \prod_{i=1}^{m} p(x_i | Pa_{x_i})$

Each of the pieces of the product is a *conditional probability distribution*.

Conditional independence

introduction:

- Two variables a, b are **independent** (written $a \perp b | \emptyset$) if: $p(a, b) = p(a)p(b)$
- Two variables a, b are **conditionally independent** given c (written $a \perp b | c$) if: $p(a, b|c) = p(a|c)p(b|c)$

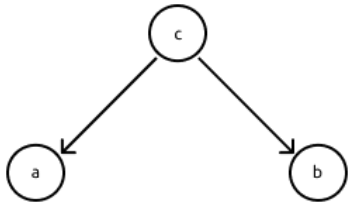
- Independence assumptions can be verified by **repeated applications of sum and product rules**
- Graphical models allow to directly verify them through the **d-separation criterion**

#1 doesn't imply #2

> Sum rule: $p(a) = \sum_b p(a, b)$
>
> product rule: $p(a, b) = p(a|b)p(b)$
>
> Bayes rule: $p(a, b) = p(a|b)p(b) = p(a|b)p(b) \Rightarrow p(a|b) = \frac{p(b|a)p(a)}{p(b)}$
>
> Independency in network can be verify by applying those rules and visually inspecting the network if know what to watch

d- separation

Tail to Tail



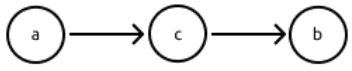
- Joint distribution: $p(a, b, c) = p(a|c)p(b|c)p(c)$
- a and b are **not independent** (written $a \top\top b|\emptyset$): $p(a, b) = \sum_c p(a|c)p(b|c)p(c) \neq p(a)p(b)$
- (is if c is observed) a and b are **conditionally independent** given c: $p(a, b|c) = \frac{p(a, b, c)}{p(c)} = p(a|c)p(b|c)$
- c is tail-to-tail with respect to to the path $a \rightarrow b$ as it is connected to the tails of the two arrows

c is connected by tails to the other two (the other are the head)

a and b are not independent

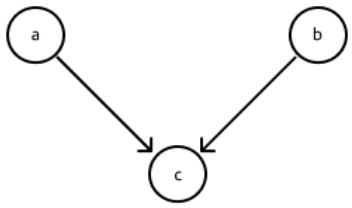
> think about c = flu, a = fever and b = cough; if $a = 1$, it increase the possibility of $b = 1$

Head to tail



- Joint distribution: $p(a, b, c) = p(b|c)p(c|a)p(a) = p(b|c)p(a|c)p(c)$
- a and b are **not independent**: $p(a, b) = p(a) \sum_c p(b|c)p(c|a) \neq p(a)p(b)$
- (is if c is observed) a and b are **conditionally independent** given c: $p(a, b|c) = \frac{p(b|c)p(a|c)p(c)}{p(c)} = p(b|c)p(a|c)$
- c is head-to-tail with respect to to the path $a \rightarrow b$ as it is connected to the head of an arrow and to the tail of the other one

Head to head



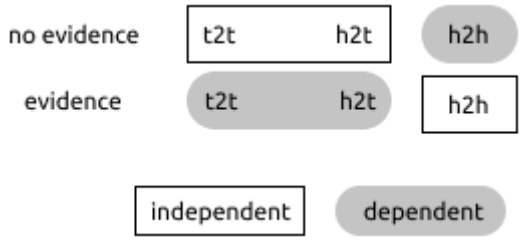
- Joint distribution: $p(a,b,c) = p(c|a,b)p(a)p(b)$
- a and b are **independent**: $p(a,b) = \sum_c p(c|a,b)p(a)p(b) = p(a)p(b)$
- (is if c is observed) a and b are **not conditionally independent** given c: $p(a,b|c) = \frac{p(c|a,b)p(a)p(b)}{p(c)} \neq p(a|c)p(b|c)$
- c is head-to-head with respect to to the path $a \rightarrow b$ as it is connected to the heads of the two arrows

> a burglar, b earthquake, c alarm: a and b become dependent if c observed (if alarm started and there is a earthquake, the burglar becomes less probable)

General Head to Head

- Let a descendant of a node x be any node which can be reached from x with a path following the direction of the arrows
- A head-to-head node c unblocks the dependency path between its parents if either itself or any of its descendants receives evidence

d-separation: basic rules summary



d-separation criterion

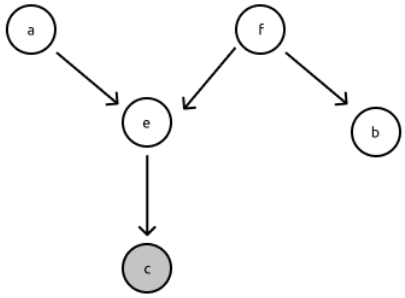
D-separation definition:

- Given a generic Bayesian network
- Given A, B, C arbitrary nonintersecting sets of nodes
- The sets A and B are *d-separated* by C ($\text{dsep}(A;B|C)$) if: all paths from any node in A to any node in B are blocked
- A **path is blocked** if it includes at least one node s.t. either:
  - the arrows on the path meet *tail-to-tail* or *head-to-tail* at the node and it is in C, or
  - the arrows on the path meet *head-to-head* at the node and neither it nor any of its descendants is in C

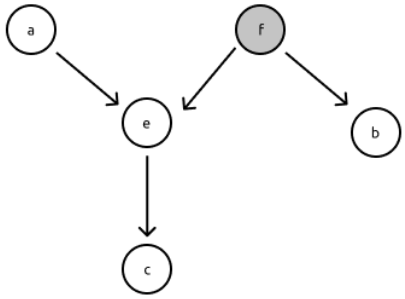
d-separation implies conditional independence: The sets A and B are independent given $C (A \perp B|C)$ if they are d-separated by C.

Examples:

- $a \top\top b|c$: Nodes a and b are not d-separated by c (between a and b there is not c):



  - Node f is *tail-to-tail* and not observed
  - Node e is *head-to-head* and its child c is observed
- $a \perp b | f$: Nodes a and b are d-separated by f (between a and b there is f):



  - Node f is *tail-to-tail* and observed
  - $\rightarrow f$ observed blocks the flow

BN independeces revisited

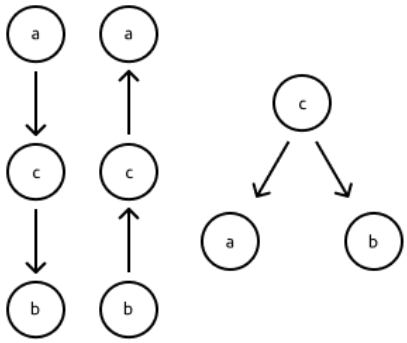
Any pairs of sets that it's separated by another set is an independence.

A BN structure G encodes a set of global (**Markov**) **independence assumptions**:
$I(G) = (A \perp B | C) : \text{dsep}(A; B | C)$

BN equivalence classes

Different BN can encode the same independencies.

All of this say that A and B are independent and become dependent if C is observed:



When you try to model a distribution using a BN you try to encode independencies in the distribution and your graphical model has to be an I-map for p.

The number of independency can make the network more or less connected, and if we connect each pair of variables then we are coding *no independency* and you are *not saving any computation* even if it's a *valid model* for any distribution.

So, we want a network as sparse as possible with less as few edges as possible.

Minimal I-maps:

- For a structure G to be an I-map for p, it does not need to encode all its independences (e.g. a fully connected graph is an I-map of any p defined over its variables)
- A minimal I-map for p is an I-map G which can't be "reduced" into a $G_0 \subset G$ (by removing edges) that is also an I-map for p.

Problem: A minimal I-map for p does not necessarily capture all the independences in p.

> You can't remove any edge

And a **perfect map** encodes all and only the independency in a distribution:

- A structure G is a perfect map (**P-map**) for p if it captures all (and only) its independences: $I(G) = I(p)$
- There exists an algorithm for finding a P-map of a distribution which is exponential in the in-degree of the P-map
- The algorithm returns an **equivalence class** rather than a single structure

Problem: Not all distributions have a P-map. Some cannot be modelled exactly by the BN formalism.

Build a BN

Because we want to model a probabilistically uncertain domain. Suggestions:

- Get together with a domain expert
- Define variables for entities that can be observed or that you can be interested in predicting (latent variables can also be sometimes useful)
- Try following causality considerations in adding edges (more interpretable and sparser networks)
- In defining probabilities for configurations (almost) never assign zero probabilities
- If data are available, use them to help in learning parameters and structure

Learning in Graphical Models

Let's start with the model given:

- We are given a dataset of examples (**training set**) $D = \{\mathbf{x}(1), \dots, \mathbf{x}(N)\}$
- each example $\mathbf{x}(i)$ is a configuration for the variables in the network. If it's a configuration with every variable is complete, else is incomplete (missing data)

We need to **estimate the parameters** of the model (conditional probability distributions) from the data: the simplest approach consists of learning the parameters *maximizing the likelihood* of the data: $\theta^{\max} = \text{argmax}_\theta p(D|\theta) = \text{argmax}_\theta L(D, \theta) \rightarrow$ we maximize the probability of the example given the parameter, and the likelihood of the data given the parameters.

ML estimation with complete data

I want to write down the entire relationship between the data and the parameters.
$p(D|\theta) = \prod_{i=1}^{N} p(\mathbf{x}(i)|\theta)$ with the examples independent given θ. We can write the probability associated to the example as $\prod_{i=1}^{N}\prod_{j=1}^{m} p(\mathbf{x}_j(i)|\text{Pa}_j(i), \theta)$ with the factorization for Bayesian Network.

Each instance of the BN is related to a different example. The parameters can be themselves decomposed with disjoint CPD parameters $\prod_{i=1}^{N}\prod_{j=1}^{m} p(\mathbf{x}_j(i)|\text{pa}_j(i), \theta_{X_j|\text{Pa}_j})$

> Conditional Probability Distribution

The parameters of each CPD can be estimated independently:

$$\theta^{\max}_{X_j|\text{Pa}_j} = \text{argmax}_{\theta_{X_j|\text{Pa}_j}} \overbrace{\prod_{i=1}^{N} p(\mathbf{x}_j(i)|\text{pa}_j(i), \theta_{X_j|\text{Pa}_j})}^{L(\theta_{X_j|\text{Pa}_j}, D)}$$

A discrete CPD $P(X|\mathbf{U})$, can be represented as a **table**, with:

- a **number of rows** equal to the number $\text{Val}(X)$ of configurations for X
- a **number of columns** equal to the number $\text{Val}(\mathbf{U})$ of configurations for its parents $\mathbf{U}$
- each table entry $\theta_{x|\mathbf{u}}$ indicating the probability of a specific configuration of $X = x$ and its parents $\mathbf{U} = \mathbf{u}$

> j is independent, all nodes are treated independently, so it's just "any node"

Replacing $p(x(i)|\text{pa}(i))$ with $\theta_{x(i)|\mathbf{u}(i)}$, the local likelihood of a single CPD becomes:

$$\begin{aligned} L(\theta_{X|\text{Pa}}, D) &= \prod_{i=1}^{N} p(x(i)|\text{pa}(i), \theta_{X|\text{Pa}_j}) \\ &= \prod_{i=1}^{N} \theta_{x(i)|\mathbf{u}(i)} \\ &= \prod_{\mathbf{u}\in\text{Val}(U)} \left[\prod_{x\in\text{Val}(X)} \theta_{x|\mathbf{u}}^{N_{\mathbf{u},x}} \right] \end{aligned}$$

where $N_{\mathbf{u},x}$ is the number of times the specific configuration $X = x$, $\mathbf{U} = \mathbf{u}$ was found in the data.

A column in the CPD table contains a multinomial distribution over values of X for a certain configuration of the parents $\mathbf{U}$.

Thus *each column should sum to one*: $\sum_x \theta_{x|\mathbf{u}} = 1$

Parameters of different columns can be estimated independently.

For each multinomial distribution, zeroing the gradient of the maximum likelihood and considering the normalization constraint, we obtain: $\theta^{\max}_{x|\mathbf{u}} = \frac{N_{\mathbf{u},x}}{\sum_x N_{\mathbf{u},x}}$

The maximum likelihood parameters are simply the fraction of times in which the specific configuration was observed in the data.

The *result* will be: what is the prob of a parameter? The fraction of time we see such parameter.

Introducing priors (MAP)

Means modelling the **posterior distribution** of the parameters given the data, **maximizing** it instead of the likelihood.

ML estimation tends to *overfit* the training set.

Configuration *not* appearing in the training set will receive *zero probability*: a common approach consists of combining ML with a prior probability on the parameters, achieving a **maximum-a-posteriori** estimate: $\theta^{\max} = \text{argmax}_\theta p(D|\theta)p(\theta)$ and maximizing $p(D|\theta)p(\theta)$ is the same as maximizing $p(\theta|D)$ because $p(D)$ is a constant, so if I maximize this, I maximize the posterior of the parameters given the data.

Dirichlet priors

The natural prior for a multinomial distribution is a Dirichlet distribution with parameters $\alpha_{x|u}$ for each possible value of x.

The resulting maximum-a-posteriori estimate is: $\theta^{\max}_{x|\mathbf{u}} = \frac{N_{\mathbf{u},x} + \alpha_{x|\mathbf{u}}}{\sum_x (N_{\mathbf{u},x} + \alpha_{x|\mathbf{u}})}$

The prior is like having observed $\alpha_{x|\mathbf{u}}$ imaginary samples with configuration $X = x, U = u$

$\alpha_{x|\mathbf{u}}$ are virtual counts of imaginary samples, considering also what there is in the prior (that can be a set of imaginary samples)

It's like computing **frequencies of configurations**, but with incomplete data we cannot compute the counts.

Incomplete data and EM for Bayesian nets

The full Bayesian approach of integrating over missing variables is often intractable in practice (summing up all alternatives possible will be expensive).

We need approximate methods to deal with the problem: we try to estimate a missing value (with the probability) given what we know, resulting in the probability of each value the missing one could become, and it will contribute with a certain fraction to some configuration and with a certain fraction in another. We use this probabilities to generate **expected counts** (probabilistic).

We now compute parameters maximizing likelihood (or posterior) with such expected counts and we iterate, improving the quality of these parameters. We iterate to improve quality of parameters (it converges to a local maximum of the likelihood)

Algorithm:

- **e-step**: Compute the expected sufficient statistics for the complete dataset, with expectation taken with respect to the joint distribution for $\mathbf{X}$ conditioned of the current value of θ and the known data D: $E_{p(\mathbf{X}|D,\theta)}[N_{ijk}] = \sum_{l=1}^{n} p(X_i(l) = x_k, \mathrm{Pa}_i(l) = \mathrm{pa}_j | \mathbf{X}_l, \theta)$
  - If $X_i(l)$ and $\mathrm{Pa}_i(l)$ are observed for $\mathbf{X}_l$, it is either zero or one
  - Otherwise, run Bayesian inference to compute probabilities from observed variables

N_{ijk} is the number of counts for the configuration where x_i takes the value x_k and the parents of i takes the value of the parents of j:

- i is the node in the network
- k is the value for that node
- j is the value of the parents
- is the l-th example
- X_l is what I know of the l-th example

> i identifies the table and j, k the coordinates in the table

If X_l is observed there is no need to do inferences, just use 0 or 1

- **m-step**: compute parameters maximizing likelihood of the complete dataset D_c (using expected counts): $\theta^* = \mathrm{argmax}_\theta p(D_c|\theta)$, which for each multinomial parameter evaluates to: $\theta^*_{ijk} = \dfrac{E_{p(\mathbf{X}|D,\theta)}[N_{ijk}]}{\sum_{k=1}^{r_i} E_{p(\mathbf{X}|D,\theta)}[N_{ijk}]}$

  > r_i is the row (of the table) we are working on

D_c is the dataset filled with expected counts.

Note: ML estimation can be replaced by *maximum a-posteriori* (MAP) estimation giving:
$\theta^*_{ijk} = \dfrac{\alpha_{ijk} + E_{p(\mathbf{X}|D,\theta)}[N_{ijk}]}{\sum_{k=1}^{r_i} \left(\alpha_{ijk} + E_{p(\mathbf{X}|D,\theta)}[N_{ijk}]\right)}$

Until now the structure was given, but what if we need to **learn the structure**?

- **constraint-based**: test conditional independencies on the data and construct a model satisfying them
- **score-based**: assign a score to each possible structure, define a search procedure looking for the structure maximizing the score
- **model-averaging**: assign a prior probability to each structure, and average prediction over all possible structures weighted by their probabilities (full Bayesian, intractable)

score-based is not monotonic

> Learning the structure model averaging is very complex because the number of structures is exponential in the number of variables

Naïve Bayes

based only on the slides, no lecture found

Naïve Bayes Classifier

The setting:

- each instance x is described by a conjunction of attribute values $\langle a_1, \ldots, a_m \rangle$
- the target function can take any value from a finite set Y
- the task is to predict the MAP target value given the instance:

$$y^* = \text{argmax}_{y_i \in Y} P(y_i|x) =$$

$$\text{argmax}_{y_i \in Y} \frac{P(a_i, \ldots, a_m|y_i)P(y_i)}{P(a_i, \ldots, a_m)} =$$

$$\text{argmax}_{y_i \in Y} P(a_i, \ldots, a_m|y_i)P(y_i)$$

> argmax is a mathematical function that you may encounter in applied machine learning. It's an operation that finds the argument that gives the maximum value from a target function.

Assumption

The **learning problem**: Class conditional probabilities $P(a_1, \ldots, a_m|y_i)$ are hard to learn, as the number of terms is equal to the number of possible instances times the number of target values.

Simplifying assumption

1. **attribute** values are assumed **independent** of each other given the target value:
   $P(a_1, \ldots, a_m|y_i) = \prod_{j=1}^{m} P(a_j|y_i)$
2. **Parameters** to be learned **reduced** to the number of possible attribute values times the number of possible target values

Classifier

Definition: $y^* = \text{argmax}_{y_i \in Y} \prod_{j=1}^{m} P(a_j|y_i)Py(y_i)$

Single distribution case

We assume all attribute values come from the same distribution $\rightarrow$ the probability of an attribute value given the class can be modelled as a multinomial distribution over the K possible values:

$P(a_j|y_i) = \sum_{k=1}^{K} \theta_{ky_i}^{z_k(a_j)}$

Parameters learning

Target priors $P(y_i)$ can be learned as the fraction of training set instances having each target value $\rightarrow$ the maximum-likelihood estimate for the parameter θ_{kc} (probability of value v_k given class c) is the fraction of times the value was observed in training examples of class c: $\theta_{kc} = \frac{N_{kc}}{N_c}$

Assume a Dirichlet prior distribution (with parameters $\alpha_{1c}, \ldots, \alpha_{Kc}$) for attribute parameters; the posterior distribution for attribute parameters is again multinomial: $\theta_{kc} = \frac{N_{kc} + \alpha_{kc}}{N_c + \alpha_c}$

Example: text classification

task description

the task is to classify documents in one of C possible classes. Each document is represented as the bag-of-words it contains (no position). Let V be the vocabulary of all possible words. A dataset of labelled documents D is available.

learning

1. compute prior probs of classes as $P(y_i) = \frac{D_i}{D}$ where D_i is the subset of training examples with class y_i
2. model attributes with a multinomial distribution with $K = |V|$ possible states (words)
3. compute the probability of the word w_k given the class c as the fraction of times the word appears in documents of class y_i, with respect to all words in documents of class c:

$$\theta_{kc} = \frac{\sum_{\mathbf{x} \in D_c} \sum_{j=1}^{|\mathbf{x}|} z_k(x[j])}{\sum_{\mathbf{x} \in D_c} |\mathbf{x}|}$$

classification

$$y^* = \text{argmax}_{y_i \in Y} \prod_{j=1}^{|\mathbf{x}|} P(x[j], y_i) P(y_i) =$$

$$\text{argmax}_{y_i \in Y} \prod_{j=1}^{|\mathbf{x}|} \prod_{k=1}^{K} \theta_{k y_i}^{z_k(x[j])} \frac{|D_i|}{|D|}$$

notes

We are making the simplifying assumption that all attribute values come from the same distribution, otherwise attributes from different distributions have to be considered separately for parameter estimation.

Maximum-likelihood and Bayesian parameter estimation

based only on the slides, no lecture found

Setting:

- Data are sampled from a **probability distribution** $p(x, y)$
- The form of the probability distribution p is known but its *parameters* are *unknown*
- There is a **training set** $D = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ of examples sampled i.i.d. according to $p(x, y)$

> **iid** = independent and identically distributed, each variable has the same probability distribution as the others and they are all independent

Task: Estimate the unknown parameters of p from training data D.

Multiclass classification setting:

- The training set can be divided into $D_1, \ldots, D_c$ subsets, one for each class (e.g. $D_i = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ contains i.i.d examples for target class y_i)
- For any new example $\mathbf{x}$ (not in training set), we compute the **posterior probability** of the class given the example and the full training set D: $P(y_i|\mathbf{x}, D) = \frac{p(\mathbf{x}|y_i, D)p(y_i|D)}{p(\mathbf{x}|D)}$

> Bayesian decision theory: compute posterior probability of class given example

Setting simplifications

- we assume $\mathbf{x}$ is independent of $D_j (j \neq i)$ given y_i and D_i
- without additional knowledge, $p(y_i|D)$ can be computed as the fraction of examples with that class in the dataset
- the normalizing factor $p(\mathbf{x}|D)$ can be computed marginalizing $p(\mathbf{x}|y_i, D_i)p(y_i|D)$ over possible classes

$\Rightarrow$ We must **estimate class-dependent parameters** θ_i for $p(\mathbf{x}|y_i, D_i)$

Maximum Likelihood vs Bayesian Estimation

Maximum Likelihood/Maximum a-posteriori estimation

It assumes *parameters* θ_i have *fixed but unknown* values; values are computed as those **maximizing** the probability of the observed examples D_i (the training set for the class). Obtained values are used to compute probability for new examples: $p(\mathbf{x}|y_i, D_i) \approx p(\mathbf{x}|\theta_i)$

Bayesian Estimation

It assumes parameters θ_i are random variables with some **known prior distribution**; *observing examples* turns prior distribution over parameters into a **posterior distribution**. Predictions for new examples are obtained integrating over all possible values for the parameters:

$p(\mathbf{x}|y_i, D_i) = \int_{\theta_i} p(\mathbf{x}, \theta_i|y_i, D_i)d\theta_i$

Maximum Likelihood

Maximum a-posteriori estimation

$\theta_i^* = \text{argmax}_{\theta_i}\, p(\theta_i|D_i, y_i) = \text{argmax}_{\theta_i}\, p(D_i, y_i|\theta_i)p(\theta_i)$

It assumes a **prior distribution** for the parameters $p(\theta_i)$ is **available**.

Maximum likelihood estimation

$\theta_i^* = \text{argmax}_{\theta_i}\, p(D_i, y_i|\theta_i)$

It **maximizes the likelihood** of the **parameters** with respect to the training samples.

No assumption about prior distributions for parameters.

Setting

- A **training data** $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of i.i.d. examples for the target class y is available
- We assume the **parameter vector** θ has a **fixed but unknown** value
- We estimate such value *maximizing its likelihood* with respect to the training data:
  $\theta^* = \text{argmax}_\theta\, p(D|\theta) = \text{argmax}_\theta \prod_{j=1}^{n} p(\mathbf{x}_j, \theta)$
- The joint **probability** over D decomposes **into a product** as examples are i.i.d (thus independent of each other given the distribution)

Maximizing log-likelihood

It is usually simpler to maximize the *logarithm* of the likelihood (monotonic):

$\theta^* = \text{argmax}_\theta \ln p(D|\theta) = \text{argmax}_\theta \sum_{j=1}^{n} \ln p(\mathbf{x}_j|\theta)$

> Monotonic = preserves the order

Necessary conditions for the maximum can be obtained zeroing the gradient with respect to θ:

$\nabla_\theta \sum_{j=1}^{n} \ln p(\mathbf{x}_j|\theta) = 0$ (points zeroing the gradient can be local or global maxima depending on the form of the distribution)

Univariate Gaussian case: unknown μ and σ^2

The **log-likelihood** is: $L = \sum_{j=1}^{n} -\frac{1}{2\sigma^2}(x_j - \mu)^2 - \frac{1}{2}\ln 2\pi\sigma^2$

The *gradient* with respect to μ is: $\frac{\delta L}{\delta \mu} = 2\sum_{j=1}^{n} -\frac{1}{2\sigma^2}(x_j - \mu)(-1) = \sum_{j=1}^{n} \frac{1}{\sigma^2}(x_j - \mu)$

Setting the **gradient to zero** gives **mean**:

$$\sum_{j=1}^{n} \frac{1}{\sigma^2}(x_j - \mu) = 0$$

$$\sum_{j=1}^{n}(x_j - \mu) = 0$$

$$\sum_{j=1}^{n} x_j = \sum_{j=1}^{n} \mu$$

$$\sum_{j=1}^{n} x_j = n\mu$$

$$\mu = \frac{1}{n}\sum_{j=1}^{n} x_j$$

The *gradient* with respect to σ^2 is:

$$\begin{aligned}\frac{\delta L}{\delta \sigma^2} &= \sum_{j=1}^{n} -(x_j - \mu)^2 \frac{\delta}{\delta \sigma^2}\frac{1}{2\sigma^2} - \frac{1}{2}\frac{1}{2\pi\sigma^2}2\pi \\ &= \sum_{j=1}^{n} -(x_j - \mu)^2 \frac{1}{2}(-1)\frac{1}{\sigma^4} - \frac{1}{2\sigma^2}\end{aligned}$$

Setting the **gradient to zero** gives **variance**:

$$\sum_{j=1}^{n} \frac{1}{2\sigma^2} = \sum_{j=1}^{n} \frac{(x_j - \mu)^2}{2\sigma^4}$$

$$\sum_{j=1}^{n} \sigma^2 = \sum_{j=1}^{n} (x_j - \mu)^2$$

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

Multivariate Gaussian case: unknown μ and Σ

The **log-likelihood** is: $\sum_{j=1}^{n} -\frac{1}{2}(\mathbf{x}_j - \mu)^t \Sigma^{-1}(\mathbf{x}_j - \mu) - \frac{1}{2}\ln(2\pi)^d|\Sigma|$

The **maximum-likelihood estimates** are: $\mu = \frac{1}{2}\sum_{j=1}^{n} \mathbf{x}_j$

and: $\Sigma = \frac{1}{n}\sum_{j=1}^{n}(\mathbf{x}_j - \mu)(\mathbf{x}_j - \mu)^t$

General Gaussian case

Maximum likelihood estimates for Gaussian parameters are simply **their empirical estimates** over the samples:

- *Gaussian mean* is the *sample mean*
- *Gaussian covariance matrix* is the *mean of sample covariances*

Bayesian Estimation

Setting:

- Assumes **parameters** θ_i are **random variables** with some known **prior distribution**

- Predictions for new examples are obtained integrating over all possible values for the parameters: $p(\mathbf{x}|y_i, D_i) = \int_{\theta_i} p(\mathbf{x}, \theta_i|y_i, D_i)d\theta_i$
- probability of $\mathbf{x}$ given each class y_i is *independent* of the other classes y_j, (for simplicity) we can again write: $p(\mathbf{x}|y_i, D_i) \rightarrow p(\mathbf{x}|D) = \int_\theta p(\mathbf{x}, \theta|D)d\theta = \int p(\mathbf{x}|\theta)p(\theta|D)d\theta$
  - where D is a dataset for a certain class y and θ are the parameters of the distribution
- $p(\mathbf{x}|\theta)$ can be easily computed (we have both form and parameters of distribution, e.g. Gaussian)
- need to **estimate** the **parameter posterior** density given the training set: $p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}$
  - $p(D)$ is a *constant independent* of θ (i.e. it will *no influence* final Bayesian *decision*)
  - if the *final probability* (not only the decision) is needed, we can compute: $p(D) = \int_\theta p(D|\theta)p(\theta)d\theta$

Univariate normal case: unknown μ, known σ^2

- Examples are drawn from: $p(x|\mu) \sim N(\mu, \sigma^2)$
- The Gaussian **mean prior** distribution is *itself normal*: $p(\mu) \sim N(\mu_0, \sigma_0^2)$
- The Gaussian **mean posterior** given the dataset *is computed* as: $p(\mu|D) = \frac{p(D|\mu)p(\mu)}{p(D)} = \alpha \prod_{j=1}^{n} p(x_j|\mu)p(\mu)$
  - where $\alpha = \frac{1}{p(D)}$ is independent of μ

A posteriori parameter density:

$$
\begin{aligned}
p(\mu|D) = \alpha \prod_{j=1}^{n} \overbrace{\frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{1}{2}\left(\frac{x_j-\mu}{\sigma}\right)^2\right]}^{p(x_j|\mu)} \overbrace{\frac{1}{\sqrt{2\pi}\sigma_0}\exp\left[-\frac{1}{2}\left(\frac{\mu-\mu_0}{\sigma_0}\right)^2\right]}^{p(\mu)} \\
= \alpha' \exp\left[-\frac{1}{2}\left(\sum_{j=1}^{n}\left(\frac{\mu-x_j}{\sigma}\right)^2 + \left(\frac{\mu-\mu_0}{\sigma_0}\right)^2\right)\right] \\
= \alpha'' \exp\left[-\frac{1}{2}\left[\left(\frac{n}{\sigma^2}+\frac{1}{\sigma_0^2}\right)\mu^2 - 2\left(\frac{1}{\sigma^2}\sum_{j=1}^{n}x_j = \frac{\mu_0}{\sigma_0^2}\right)\mu\right]\right]
\end{aligned}
$$

Normal distribution:

$$p(\mu|D) = \frac{1}{\sqrt{2\pi}\sigma_n}\exp\left[-\frac{1}{2}\left(\frac{\mu-\mu_n}{\sigma_n}\right)^2\right]$$

Recovering mean and variance

$$
\begin{gathered}
\left(\frac{n}{\sigma^2}+\frac{1}{\sigma_0^2}\right)\mu^2 - 2\left(\frac{1}{\sigma^2}\sum_{j=1}^{n}x_j + \frac{\mu_0}{\sigma_0^2}\right)\mu + \alpha''' = \\
\left(\frac{\mu-\mu_n}{\sigma_n}\right)^2 = \frac{1}{\sigma_n^2}\mu^2 - 2\frac{\mu_n}{\sigma_n^2}\mu + \frac{\mu_n^2}{\sigma_n^2}
\end{gathered}
$$

Solving for μ_n and σ_n^2 we obtain

$$\mu_n = \left(\frac{n\sigma_0^2}{n\sigma_0^2 + \sigma^2} \right) \hat{\mu}_n + \frac{\sigma^2}{n\sigma_0^2 + \sigma^2} \mu_0$$

$$\text{and}$$

$$\sigma_n^2 = \frac{\sigma_0^2 \sigma^2}{n\sigma_0^2 + \sigma^2}$$

where $\hat{\mu}_n$ is the *sample mean*: $\hat{\mu}_n = \frac{1}{n} \sum_{j=1}^{n} x_j$

Interpreting the posterior

- The **mean** is a *linear combination* of the *prior* (μ_0) and *sample means* ($\hat{\mu}_n$)
- The *more training examples* (n) are seen, the *more the sample mean* dominates *over prior* mean (unless $\sigma_0^2 = 0$)
- The more training examples (n) are seen, the more the variance decreases making the *distribution sharply peaked* over its mean: $\lim_{n \to \infty} \frac{\sigma_0^2 \sigma^2}{n\sigma_0^2 + \sigma^2} = \lim_{n \to \infty} \frac{\sigma^2}{n} = 0$

Computing the class conditional density

$$p(x|D) = \int p(x|\mu) p(\mu|D) d\mu$$

$$= \int \frac{1}{\sqrt{2\pi}\sigma} \exp \left[-\frac{1}{2} \left(\frac{x - \mu}{\sigma} \right)^2 \right] \frac{1}{\sqrt{2\pi}\sigma_n} \exp \left[-\frac{1}{2} \left(\frac{\mu - \mu_n}{\sigma_n} \right)^2 \right] d\mu$$

$$\sim N(\mu_n, \sigma^2 + \sigma_n^2)$$

note

The probability of x given the dataset for the class is a Gaussian with:

- *mean* equal to the posterior mean
- *variance* equal to the sum of the known variance (σ^2) and an additional variance (σ_n^2) due to the uncertainty on the mean

Random info (?)

Multivariate normal case: unknown μ, known Σ

$$\begin{aligned} p(\mathbf{x}|\mu) &\sim N(\mu, \Sigma) \\ \Rightarrow p(\mu) &\sim N(\mu_0, \Sigma_0) \\ \Rightarrow p(\mu|D) &\sim N(\mu_n, \Sigma_n) \\ \Rightarrow p(\mathbf{x}|D) &\sim N(\mu_n, \Sigma + \Sigma_n) \end{aligned}$$

> wut

Sufficient Statistics

Definition:

- Any function on a set of samples D is a **statistic**
- A statistic $\mathbf{s} = \phi(D)$ is **sufficient** for some parameters θ if: $P(D|\mathbf{s}, \theta) = P(D|\mathbf{s})$
- If θ is a random variable, a sufficient statistic contains *all relevant information* D has for estimating it:
  $P(D|\mathbf{s}, \theta) = \frac{p(D|\theta, \mathbf{s})p(\theta|\mathbf{s})}{p(D|\mathbf{s})} = p(\theta|\mathbf{s})$

Use:

- A sufficient statistic allows to **compress** a sample D into (possibly few) values
- **Sample mean** and **covariance** are sufficient statistics **for true mean** and **covariance** of the **Gaussian** distribution

Conjugate priors

Definition:

Given a likelihood function $p(x|\theta)$and given a prior distribution $p(\theta)$: $p(\theta)$ is a **conjugate prior** for $p(x|\theta)$ if the *posterior* distribution $p(\theta|x)$ is in the *same family* as the *prior* $p(\theta)$

Examples

Likelihood	**Parameters**	**Conjugate prior**
Binomial	p (probability)	Beta
Multinomial	$\mathbf{p}$ (probability vector)	Dirichlet
Normal	μ (mean)	Normal
Multivariate normal	$\boldsymbol{\mu}_i$ (mean vector)	Normal

Bernoulli distribution

Setting:

- Boolean event: $x = 1$ for success, $x = 0$ for failure
- Parameters: $\theta = \text{probability of success}$
- Probability mass function $P(x|\theta) = \theta^x(1-\theta)^{1-x}$
- Beta conjugate prior: $P(\theta|\psi) = P(\theta|\alpha_h, \alpha_t) = \frac{\Gamma(\alpha)}{\Gamma(\alpha_h)\Gamma(\alpha_t)}\theta^{\alpha_h - 1}(1-\theta)^{\alpha_t - 1}$

> skipped example with the coin toss (slides 29-30/52 of "Maximum-likelihood and Bayesian parameter estimation")

Interpreting priors:

- Our prior knowledge is encoded as a number $\alpha = \alpha_h + \alpha_t$ of imaginary experiments
- we assume α_h times we observed heads (in the coin toss example)
- α is called equivalent sample size
- $\alpha \to 0$ reduces estimation to the classical ML approach (frequentist)

Multinomial Distribution

Setting:

- Categorical event with r states $x \in \{x^1, \ldots, x^r\}$ (e.g. tossing a six-faced dice)
- One-hot encoding $\mathbf{z}(x) = [z_1(x), \ldots, z_r(x)]$ with $z_k(x) = 1$ if $x = x^k$, 0 otherwise
- Parameters: $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_r]$ probability of each state
- Probability mass function $P(x|\boldsymbol{\theta}) = \prod_{k=1}^{r} \theta_k^{z_k(x)}$
- Dirichlet conjugate prior: $P(\boldsymbol{\theta}|\psi) = P(\boldsymbol{\theta}|\alpha_1, \ldots, \alpha_r) = \frac{\Gamma(\alpha)}{\prod_{k=1}^{r} \Gamma(\alpha_k)} \prod_{k=1}^{r} \theta_k^{\alpha_k - 1}$

> skipping the example (slides 33-34/52 of “Maximum-likelihood and Bayesian parameter estimation”)