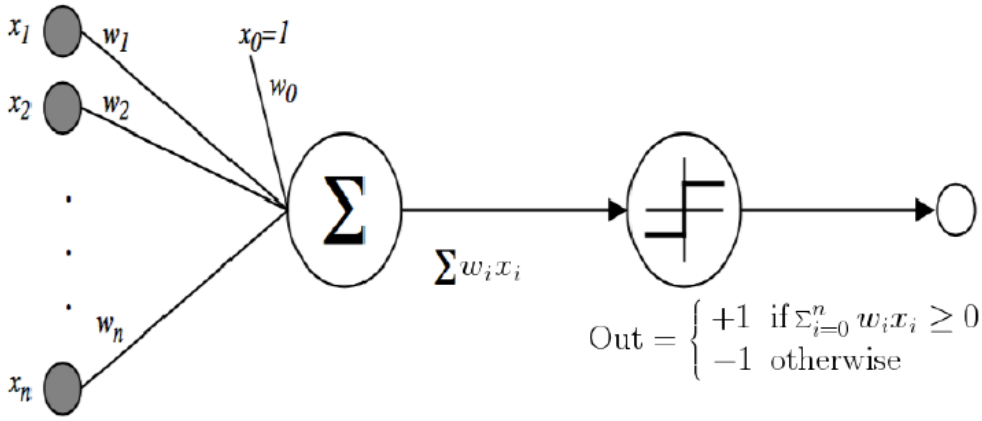
Lecture5 NeuralNet

Aims

This lecture will enable you to describe and reproduce machine learning approaches to the problem of neural (network) learning. Following it you should be able to:

- describe Perceptrons and how to train them
- relate neural learning to optimization in machine learning
- outline the problem of neural learning
- derive the method of gradient descent for linear models
- describe the problem of non-linear models with neural networks
- outline the method of back-propagation training of a multi-layer perceptron neural network
- describe the application of neural learning for classification
- describe some issues arising when training deep neural networks

Perceptron



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

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

Key idea:

Learning is "finding a good set of weights"

Perceptron learning is simply an iterative weight-update scheme:

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

where the weight update Δw_i depends only on *misclassified* examples and is modulated by a "smoothing" parameter η typically referred to as the "learning rate".

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

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

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

Algorithm Perceptron(D, η) // perceptron training for linear classification

Input: labelled training data D in homogeneous coordinates; learning rate η.
Output: weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

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

Unfortunately, as a linear classifier perceptrons are limited in expressive power

So some functions not representable

- e.g., not linearly separable

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

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

- *multilayer perceptrons*, the classic "neural network"

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

Gradient Descent

To understand, consider simpler *linear unit*, where

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

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

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

Where D is set of training examples

Gradient

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

Gradient vector gives direction of *steepest increase* in error E

Negative of the gradient, i.e., *steepest decrease*, is what we want

Training rule:

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

i.e.,

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

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

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

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

Initialize each w_i to some small random value

Until the termination condition is met, Do

  Initialize each Δw_i to zero

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

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

    For each linear unit weight w_i

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

  For each linear unit weight w_i

    $w_i \leftarrow w_i + \Delta w_i$

Batch mode Gradient Descent:
Do until satisfied

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

Incremental mode Gradient Descent:
Do until satisfied

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

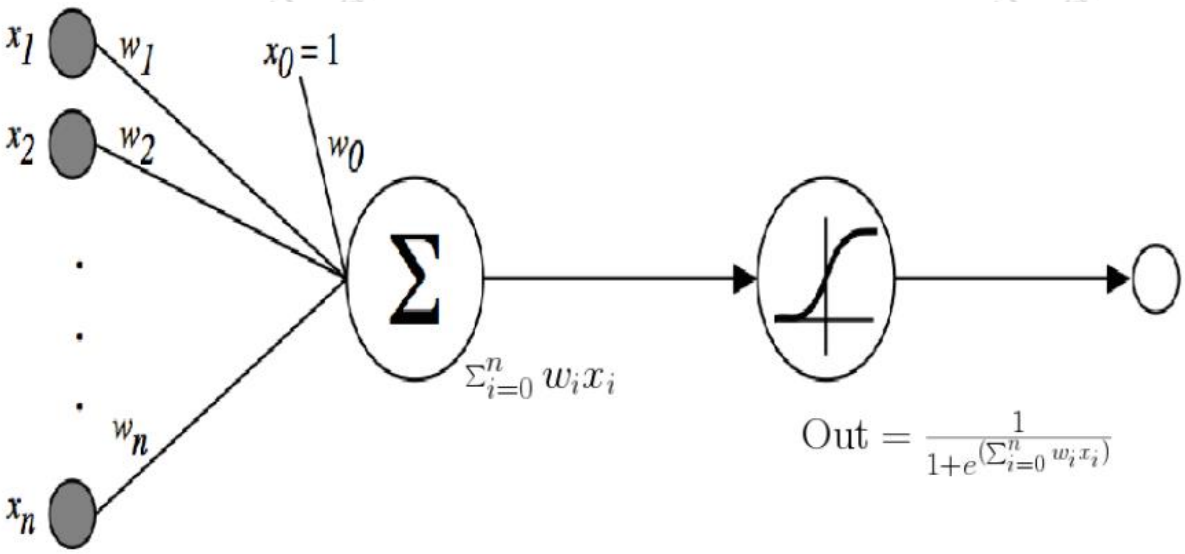
Batch:

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

Incremental:

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

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



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

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

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

We can derive gradient descent rules to train

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

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

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

But we know:

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

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

So:

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

Initialize all weights to small random numbers.

Until satisfied, Do

For each training example, Do

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

For each output unit k

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

For each hidden unit h

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

Update each network weight w_{ji}

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

where

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

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

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

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

- Stochastic gradient descent using "mini-batches"

Models can be very complex

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

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

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

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

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

Deep Learning: Regularization

Deep networks can have millions or billions or parameters.

Hard to train, prone to overfit.

What techniques can help ?

Example: dropout

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

Neural networks for classification

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

Output ranges from 0 to 1

Example: binary classification

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

Questions:

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

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

If we take the output $o(\mathbf{x})$ as the *probability* of the class of $\mathbf{x}$ being 1, the preferred loss function is the *cross-entropy*

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

where:

$t_d \in \{0,1\}$ is the class label for training example d, and o_d is the output of the sigmoid unit, interpreted as the probability of the class of training example d being 1.

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

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

Solution: use activation functions that are non-saturating., e.g., "Rectified Linear Unit" or ReLu, defined as $f(x) = \max(0, x)$.

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

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

Derivative of a ReLu is simply

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

Lecture6 KernelMethods

Aims

This lecture will develop your understanding of kernel methods in machine learning. Following it you should be able to:

- describe learning with the dual perceptron
- outline the idea of learning in a dual space
- describe the concept of maximising the margin in linear classification
- outline the typical loss function for maximising the margin
- describe the method of support vector machines (SVMs)
- describe the concept of kernel functions
- outline the idea of using a kernel in a learning algorithm
- outline non-linear classification with kernel methods

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

Linear classifiers in dual form

Every time an example $\mathbf{x}_i$ is misclassified, add $y_i\mathbf{x}_i$ to the weight vector.

- After training has completed, each example has been misclassified zero or more times. Denoting this number as α_i for example $\mathbf{x}_i$, the weight vector can be expressed as

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

- In the dual, instance-based view of linear classification we are learning instance weights α_i rather than feature weights w_j. An instance $\mathbf{x}$ is classified as

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

- During training, the only information needed about the training data is all pairwise dot products: the n-by-n matrix $\mathbf{G} = \mathbf{X}\mathbf{X}^{\mathrm{T}}$ containing these dot products is called the **Gram matrix**.

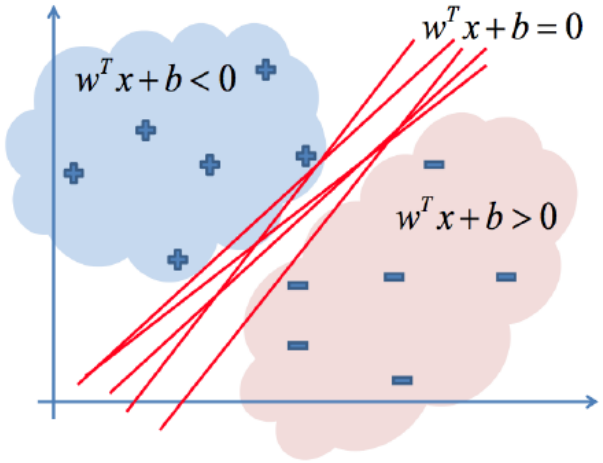
Algorithm DualPerceptron(D) // perceptron training in dual form

Input: labelled training data D in homogeneous coordinates

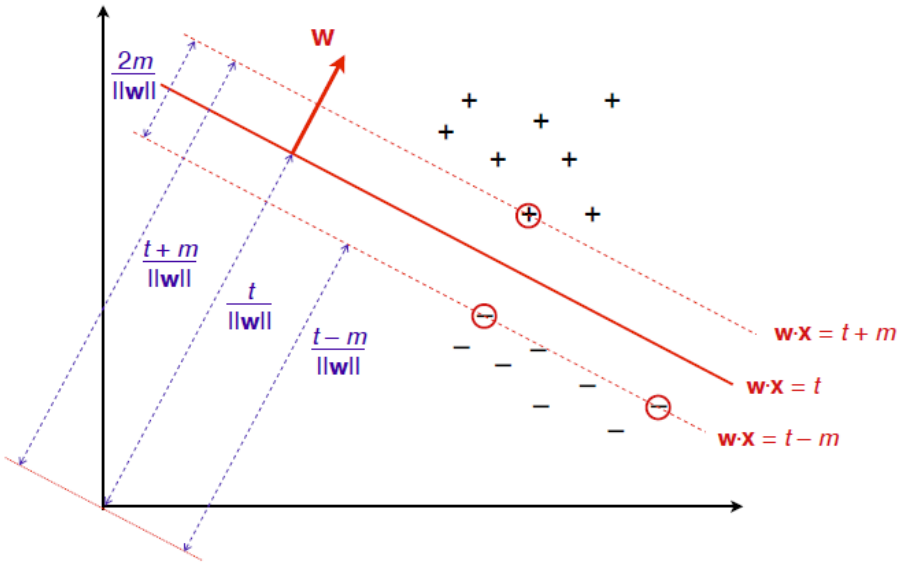
Output: coefficients α_i defining weight vector $\mathbf{w} = \sum_{i=1}^{|D|} \alpha_i y_i \mathbf{x}_i$

```
1  α_i ← 0 for 1 ≤ i ≤ |D|
2  converged←false
3  while converged = false do
4      converged←true
5      for i = 1 to |D| do
6          if y_i Σ_{j=1}^{|D|} α_j y_j x_i · x_j ≤ 0 then
7              α_i ← α_i + 1
8              converged←false
9          end
10     end
11 end
```

- Many possible linear decision boundaries: which one to choose ?



Support vector machine



The geometry of a support vector classifier. The circled data points are the support vectors, which are the training examples nearest to the decision boundary. The support vector machine finds the decision boundary that maximises the margin $m/||\mathbf{w}||$.

$$\mathbf{w}^*, t^* = \arg\min_{\mathbf{w},t} \frac{1}{2}||\mathbf{w}||^2 \qquad \text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1, 1 \leq i \leq n$$

Adding the constraints with multipliers α_i for each training example gives the Lagrange function

$$\begin{aligned}
\Lambda(\mathbf{w}, t, \alpha_1, \ldots, \alpha_n) &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\mathbf{w}\cdot\mathbf{x}_i - t) - 1) \\
&= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i y_i (\mathbf{w}\cdot\mathbf{x}_i) + \sum_{i=1}^{n} \alpha_i y_i t + \sum_{i=1}^{n} \alpha_i \\
&= \frac{1}{2}\mathbf{w}\cdot\mathbf{w} - \mathbf{w}\cdot\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) + t\left(\sum_{i=1}^{n} \alpha_i y_i\right) + \sum_{i=1}^{n} \alpha_i
\end{aligned}$$

- By taking the partial derivative of the Lagrange function with respect to t and setting it to 0 we find $\sum_{i=1}^{n} \alpha_i y_i = 0$.
- Similarly, by taking the partial derivative of the Lagrange function with respect to $\mathbf{w}$ and setting to 0 we obtain $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$ – the same expression as we derived for the perceptron.

These expressions allow us to eliminate $\mathbf{w}$ and t and lead to the dual Lagrangian

$$\begin{aligned}
\Lambda(\alpha_1, \ldots, \alpha_n) &= -\frac{1}{2}\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right)\cdot\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) + \sum_{i=1}^{n} \alpha_i \\
&= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i
\end{aligned}$$

The dual optimisation problem for support vector machines is to maximise the dual Lagrangian under positivity constraints and one equality constraint:

$$\alpha_1^*, \ldots, \alpha_n^* = \arg\max_{\alpha_1,\ldots,\alpha_n} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^{n}\alpha_i$$

$$\text{subject to } \alpha_i \geq 0, \;\; 1 \leq i \leq n \;\text{ and }\; \sum_{i=1}^{n}\alpha_i y_i = 0$$

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix} \qquad \mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \end{pmatrix}$$

The matrix $\mathbf{X}'$ on the right incorporates the class labels; i.e., the rows are $y_i\mathbf{x}_i$. The Gram matrix is (without and with class labels):

$$\mathbf{X}\mathbf{X}^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & -5 \\ 3 & 5 & -3 \\ -5 & -3 & 5 \end{pmatrix} \qquad \mathbf{X}'\mathbf{X}'^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & 5 \\ 3 & 5 & 3 \\ 5 & 3 & 5 \end{pmatrix}$$

The dual optimisation problem is thus

$$\arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(5\alpha_1^2 + 3\alpha_1\alpha_2 + 5\alpha_1\alpha_3 + 3\alpha_2\alpha_1 + 5\alpha_2^2 + 3\alpha_2\alpha_3 + 5\alpha_3\alpha_1 + 3\alpha_3\alpha_2 + 5\alpha_3^2\right) + \alpha_1 +$$

$$= \arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1\alpha_3 + 5\alpha_2^2 + 6\alpha_2\alpha_3 + 5\alpha_3^2\right) + \alpha_1 + \alpha_2 + \alpha_3$$

subject to $\alpha_1 \geq 0, \alpha_2 \geq 0, \alpha_3 \geq 0$ and $-\alpha_1 - \alpha_2 + \alpha_3 = 0$.

- Using the equality constraint we can eliminate one of the variables, say α_3, and simplify the objective function to

$$\arg\max_{\alpha_1,\alpha_2,\alpha_3} -\frac{1}{2}\left(20\alpha_1^2 + 32\alpha_1\alpha_2 + 16\alpha_2^2\right) + 2\alpha_1 + 2\alpha_2$$

- Setting partial derivatives to 0 we obtain $-20\alpha_1 - 16\alpha_2 + 2 = 0$ and $-16\alpha_1 - 16\alpha_2 + 2 = 0$ (notice that, because the objective function is quadratic, these equations are guaranteed to be linear).
- We therefore obtain the solution $\alpha_1 = 0$ and $\alpha_2 = \alpha_3 = 1/8$. We then have $\mathbf{w} = 1/8(\mathbf{x}_3 - \mathbf{x}_2) = \begin{pmatrix} 0 \\ -1/2 \end{pmatrix}$, resulting in a margin of $1/||\mathbf{w}|| = 2$.
- Finally, t can be obtained from any support vector, say $\mathbf{x}_2$, since $y_2(\mathbf{w} \cdot \mathbf{x}_2 - t) = 1$; this gives $-1 \cdot (-1 - t) = 1$, hence $t = 0$.

Allowing margin errors

The idea is to introduce *slack variables* ξ_i, one for each example, which allow some of them to be inside the margin or even at the wrong side of the decision boundary. We will call these *margin errors*. Thus, we change the constraints to $\mathbf{w} \cdot \mathbf{x}_i - t \geq 1 - \xi_i$ and add the sum of all slack variables to the objective function to be minimised, resulting in the following *soft margin* optimisation problem:

$$\mathbf{w}^*, t^*, \xi_i^* = \arg\min_{\mathbf{w},t,\xi_i} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$
$$\text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n$$

The Lagrange function is then as follows:

$$\begin{aligned}\Lambda(\mathbf{w}, t, \xi_i, \alpha_i, \beta_i) &= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(\mathbf{w}\cdot\mathbf{x}_i - t) - (1-\xi_i)) - \sum_{i=1}^{n}\beta_i\xi_i \\ &= \Lambda(\mathbf{w}, t, \alpha_i) + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i\end{aligned}$$

What is the significance of the upper bound C on the α_i multipliers?

- Since $C - \alpha_i - \beta_i = 0$ for all i, $\alpha_i = C$ implies $\beta_i = 0$. The β_i multipliers come from the $\xi_i \geq 0$ constraint, and a multiplier of 0 means that the lower bound is not reached, i.e., $\xi_i > 0$ (analogous to the fact that $\alpha_j = 0$ means that $\mathbf{x}_j$ is not a support vector and hence $\mathbf{w}\cdot\mathbf{x}_j - t > 1$).
- In other words, a solution to the soft margin optimisation problem in dual form divides the training examples into three cases:

  $\alpha_i = 0$ these are outside or on the margin;
  $0 < \alpha_i < C$ these are the support vectors on the margin;
  $\alpha_i = C$ these are on or inside the margin.

- Notice that we still have $\mathbf{w} = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i$, and so both second and third case examples participate in spanning the decision boundary.

Kernel trick

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$$

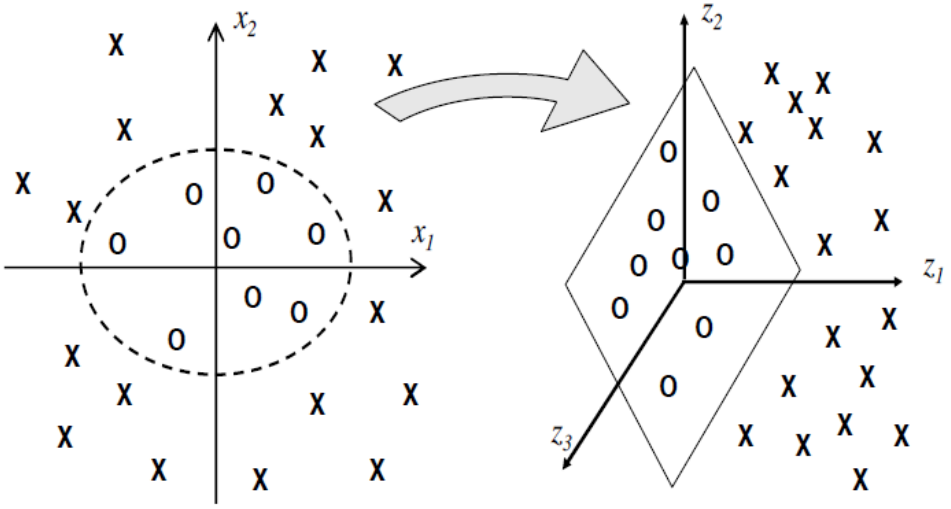


Figure by Avrim Blum, CS Dept, CMU.

$$\phi(\mathbf{x}_i) = \left(x_i^2, y_i^2, \sqrt{2}x_iy_i\right) \qquad \phi(\mathbf{x}_j) = \left(x_j^2, y_j^2, \sqrt{2}x_jy_j\right)$$

$$\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) = x_i^2x_j^2 + y_i^2y_j^2 + 2x_ix_jy_iy_j = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$$

Algorithm KernelPerceptron(D, η) // perceptron training algorithm using a ker
Input: labelled training data D in homogeneous coordinates, plus
kernel function κ
Output: coefficients α_i defining non-linear decision boundary

```
1  α_i ← 0 for 1 ≤ i ≤ |D|
2  converged←false
3  while converged = false do
4      converged←true
5      for i = 1 to |D| do
6          if y_i Σ_{j=1}^{|D|} α_j y_j κ(x_i, x_j) ≤ 0 then
7              α_i ← α_i + 1
8              converged←false
9          end
10     end
11 end
```

1. Linear Kernel

The Linear kernel is the simplest kernel function. It is given by the inner product <x,y> plus an optional constant c. Kernel algorithms using a linear kernel are often equivalent to their non-kernel counterparts, i.e. KPCAwith linear kernel is the same as standard PCA.

$$k(x, y) = x^T y + c$$

2. Polynomial Kernel

The Polynomial kernel is a non-stationary kernel. Polynomial kernels are well suited for problems where all the training data is normalized.

$$k(x, y) = (\alpha x^T y + c)^d$$

Adjustable parameters are the slope alpha, the constant term c and the polynomial degree d.

3. Gaussian Kernel

The Gaussian kernel is an example of radial basis function kernel.

$$k(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

Alternatively, it could also be implemented using

$$k(x, y) = \exp\left(-\gamma\|x - y\|^2\right)$$

The adjustable parameter sigma plays a major role in the performance of the kernel, and should be carefully tuned to the problem at hand. If overestimated, the exponential will behave almost linearly and the higher-dimensional projection will start to lose its non-linear power. In the other hand, if underestimated, the function will lack regularization and the decision boundary will be highly sensitive to noise in training data.

4. Exponential Kernel

The exponential kernel is closely related to the Gaussian kernel, with only the square of the norm left out. It is also a radial basis function kernel.

$$k(x, y) = \exp\left(-\frac{\|x - y\|}{2\sigma^2}\right)$$

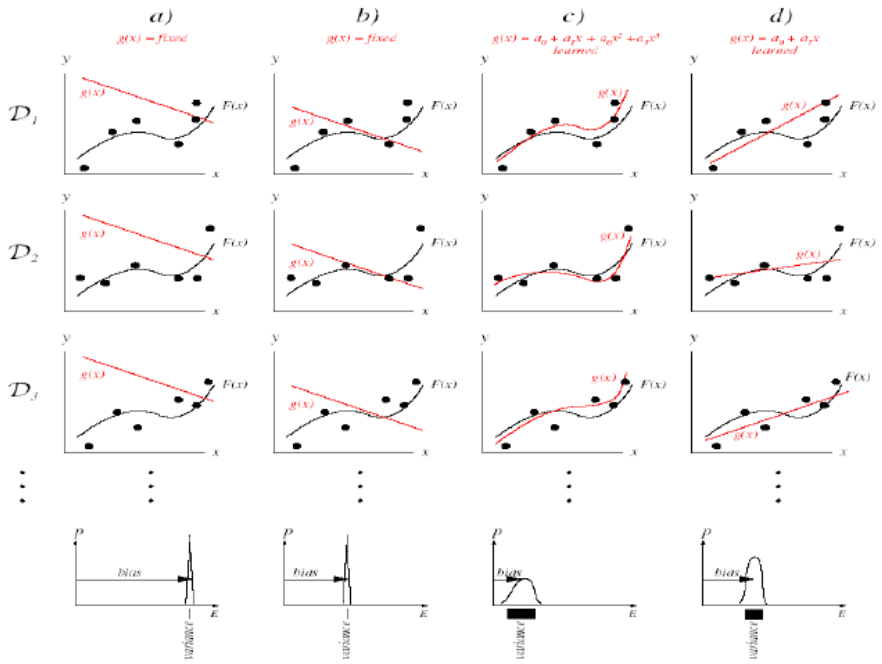
Lecture7 Ensemble Learning

Aims

This lecture will develop your understanding of ensemble methods in machine learning, based on analyses and algorithms covered previously. Following it you should be able to:

- describe the framework of the bias-variance decomposition and some of its practical implications
- describe how ensembles might be used to address the bias and variance components of error
- outline the concept of the stability of a learning algorithm
- describe the ensemble methods of bagging, random forests and boosting
- compare the operation of these methods in terms of the bias and variance components of error

- Assume we have an infinite number of classifiers built from different training sets all of the same size:
  - The *bias* of a learning scheme is the expected error due to the mismatch between the learner's hypothesis space and the space of target concepts
  - The *variance* of a learning scheme is the expected error due to differences in the training sets used
  - Total expected error $\approx$ bias^2 + variance



Training-set error is observed to be high compared to human-level – why ?

Bias is too high – solution: move to a more expressive (lower bias) model

Training-set error is observed to be similar to human-level, but validation set error is high compared to human-level – why ?

Variance is too high – solution: get more data (!), try regularization, ensembles, move to a different model architecture

These scenarios are often found in applications of deep learning[2].

Stability

- for a given data distribution $\mathcal{D}$
- train algorithm L on training sets S_1, S_2 sampled from $\mathcal{D}$
- expect that the model from L should be the same (or very similar) on both S_1 and S_2
- if so, we say that L is a *stable* learning algorithm
- otherwise it is unstable
- typical stable algorithm: kNN (for some k)
- typical unstable algorithm: decision-tree learning

Turney, P. *"Bias and the Quantification of Stability"*

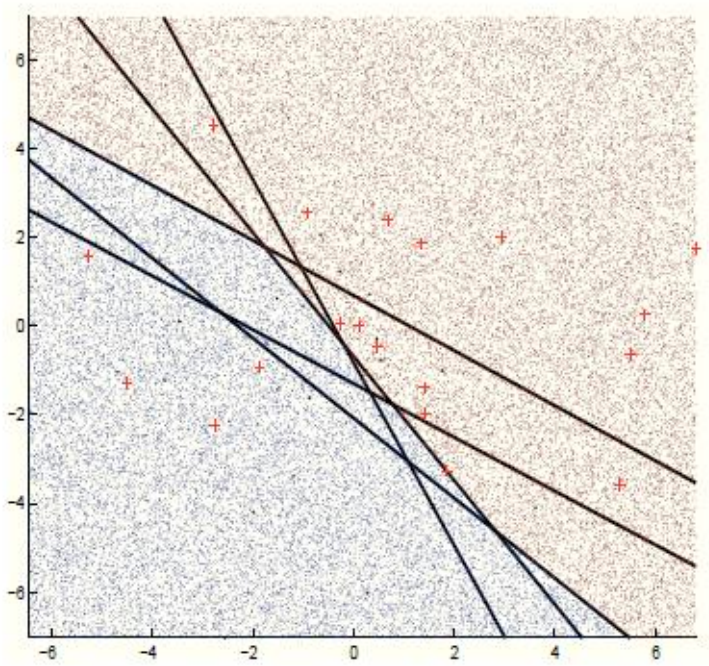
- stable algorithms typically have high bias
- unstable algorithms typically have high variance
- BUT: take care to consider effect of parameters, e.g., in kNN
  - 1NN perfectly separates training data, so low bias but high variance
  - By increasing the number of neighbours k we increase bias and decrease variance (what happens when $k = n$?)
  - Every test instance will have the same number of neighbours, and the class probability vectors will all be the same !

Algorithm Bagging$(D, T, \mathcal{A})$ // train ensemble from bootstrap samples

Input: dataset D; ensemble size T; learning algorithm $\mathcal{A}$.

Output: set of models; predictions to be combined by voting or averaging.

```
for t = 1 to T do
    bootstrap sample D_t from D by sampling |D| examples with replacement
    run A on D_t to produce a model M_t
end
return {M_t | 1 ≤ t ≤ T}
```



The news is not all good:

- when we bag a model, any simple structure is lost
- this is because a bagged tree is no longer a tree ...
- ... but a forest
- although bagged trees can be mapped back to a single tree ...
- ... this reduces claim to comprehensibility

- *stable* models like nearest neighbour not very affected by bagging
- *unstable* models like trees most affected by bagging
- usually, their design for interpretability (bias) leads to instability
- more recently, *random forests* (see Breiman's web-site)

Algorithm RandomForest(D, T, d) // train ensemble of randomized trees
Input: data set D; ensemble size T; subspace dimension d.
Output: set of models; predictions to be combined by voting or averaging.

```
for t = 1 to T do
    bootstrap sample D_t from D by sampling |D| examples with replacement
    select d features at random and reduce dimensionality of D_t accordingly
    train a tree model M_t on D_t without pruning
end
return {M_t | 1 ≤ t ≤ T}
```

- advantage: forces more diversity among trees in ensemble
- advantage: less time to train since only consider a subset of features

BOOSTING

- New model is encouraged to become "expert" for instances classified incorrectly by earlier models

- Learner produces a binary $[-1, +1]$ classifier h with error rate $\epsilon < 0.5$.
- In some sense h is "useful", i.e., better than random !
- **strong** learner if $\epsilon < 0.5$ and ϵ "close" to zero.
- **weak** learner if $\epsilon < 0.5$ and ϵ "close" to 0.5.

Simple boosting using three classifiers in an ensemble:

- h_1 is a weak learner
- dataset used to train h_2 is maximally *informative* wrt h_1
- h_3 learns on what h_1 and h_2 disagree about
- for prediction on instance x:
  - if h_1 and h_2 agree, use that label (probably correct)
  - otherwise use h_3 (probably neither h_1 or h_2 are correct)

Algorithm Boosting$(D, T, \mathcal{A})$ // train binary classifier ensemble, reweighting datasets

Input: data set D; ensemble size T; learning algorithm $\mathcal{A}$
Output: weighted ensemble of models

$w_{1i} \leftarrow 1/|D|$ for all $x_i \in D$
for $t = 1$ to T **do**
  run $\mathcal{A}$ on D with weights w_{ti} to produce a model M_t
  calculate weighted error ϵ_t
  $\alpha_t \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$
  $w_{(t+1)i} \leftarrow \frac{w_{ti}}{2\epsilon_t}$ for misclassified instances $x_i \in D$
  $w_{(t+1)j} \leftarrow \frac{w_{tj}}{2(1-\epsilon_t)}$ for correctly classified instances $x_j \in D$
end
return $M(x) = \sum_{t=1}^{T} \alpha_t M_t(x)$

Stacking

- So far, ensembles where base learners all use same algorithm
- But what if we want to combine outputs of different algorithms ?
- Also, what if the combining method could be tuned from data ?
- "Stacked generalization" or stacking
- Uses *meta learner* instead of voting to combine predictions of base learners
  - Predictions of base learners (level-0 models) are used as input for meta learner (level-1 model)
- Each base learners considered a feature, with value its output $\hat{y}$ on instance x
- But predictions on training data can't be used to generate data for level-1 model!
  - So a cross-validation-like scheme is employed

Important points to remember

Low-bias models tend to have high variance, and *vice versa*.

Bagging is predominantly a variance-reduction technique, while boosting is primarily a bias-reduction technique.

This explains why bagging is often used in combination with high-variance models such as tree models (as in Random Forests), whereas boosting is typically used with high-bias models such as linear classifiers or univariate decision trees (also called *decision stumps*).

Lecture8 Unsupervised

Aims

This lecture will develop your understanding of unsupervised learning methods. Following it you should be able to:

- compare supervised with unsupervised learning
- describe the problem of dimensionality reduction
- outline the method of Principal Component Analysis
- describe the problem of unsupervised learning
- describe k-means clustering
- outline the role of the EM algorithm in k-means clustering
- describe hierarchical clustering
- describe methods of evaluation for unsupervised learning
- outline semi-supervised learning

Supervised learning — classes are *known* and need a "definition", in terms of the data. Methods are known as: classification, discriminant analysis, class prediction, supervised pattern recognition.

Unsupervised learning — classes are initially *unknown* and need to be "discovered" with their definitions from the data. Methods are known as: cluster analysis, class discovery, unsupervised pattern recognition.

Why do we need unsupervised learning ?

- most of the world's data is *unlabelled*
- getting a human to label data is often
  - difficult (what are the classes ?)
  - time-consuming (labelling requires thinking)
  - expensive (see above)
  - error-prone (mistakes, ambiguity)
- in principle, can use any feature as the "label"
- unfortunately, often the class is not a known feature

What is unsupervised learning good for ?

- simplifying a problem, e.g., by dimensionality reduction
- exploratory data analysis, e.g., with visualization
- data transformation to simplify a classification problem
- to group data instances into subsets
- to discover structure, like hierarchies of subconcepts
- to learn new "features" for later use in classification
- to track "concept drift" over time
- to learn generative models for images, text, video, speech, etc.

PCA Algorithm

This algorithm can be presented in several ways. Here are the basic steps in terms of the variance reduction idea:

1. take the data as an $n \times m$ matrix $\mathbf{X}$
2. "centre" the data by subtracting the mean of each column
3. construct covariance matrix $\mathbf{C}$ from centred matrix
4. compute eigenvector matrix $\mathbf{V}$ (rotation) and eigenvalue matrix $\mathbf{S}$ (scaling) such that $\mathbf{V}^{-1}\mathbf{C}\mathbf{V} = \mathbf{S}$, and $\mathbf{S}$ is a diagonal $m \times m$ matrix
5. sort columns of $\mathbf{S}$ in decreasing order (decreasing variance)
6. remove columns of $\mathbf{S}$ below some minimum threshold

- PCA complexity is cubic in number of original features
- this is not feasible for high-dimensional datasets

- PCA will transform original features to new space
- every new feature is a linear combination of original features
- aim for new dimensions to maximise variance
- order by decreasing variance and remove those below a threshold
- this reduces dimensionality
- algorithm applies matrix operations to translate, rotate and scale

Cluster analysis

Clustering algorithms form two broad categories: **hierarchical methods** and **partitioning methods**.

Algorithm k-means

/* feature-vector matrix $M(ij)$ is given */

1. Start with an arbitrary partition P of N into k clusters
2. for each element i and cluster $j \neq P(i)$ let E_P^{ij} be the cost of a solution in which i is moved to j:
   1. if $E_P^{i^*j^*} = min_{ij} E_P^{ij} < E_P$ then move i^* to cluster j^* and repeat step 2 else halt.

How many clusters ?

Many methods of estimating the "correct" number of clusters have been proposed. Here we mention two using some clustering criterion (such as"dispersion"i.e.,sum of squared distances from each point to the cluster centroid).

- compare value for single cluster to sum of values for breaking cluster into two
- if there is a "significant" reduction then keep two clusters[1]
- formalise "elbow" detection
- Gap statistic[2]
- compares graph of within-cluster dispersion against k to a random reference value

EM for Estimating k Means

Given:

- Instances from X generated by mixture of k Gaussian distributions
- Unknown means $\langle \mu_1, \ldots, \mu_k \rangle$ of the k Gaussians
- Don't know which instance x_i was generated by which Gaussian

Determine:

- Maximum likelihood estimates of $\langle \mu_1, \ldots, \mu_k \rangle$

Initialise: Pick random initial $h = \langle \mu_1, \mu_2 \rangle$

Iterate:

E step:

Calculate expected value $E[z_{ij}]$ of each hidden variable z_{ij}, *assuming* current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds:

$$
\begin{aligned}
E[z_{ij}] &= \frac{p(x = x_i | \mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)} \\
&= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}}
\end{aligned}
$$

M step:

Calculate new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, *assuming* value taken on by each hidden variable z_{ij} is the expected value $E[z_{ij}]$ calculated before.
Replace $h = \langle \mu_1, \mu_2 \rangle$ by $h' = \langle \mu'_1, \mu'_2 \rangle$.

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}] \quad x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

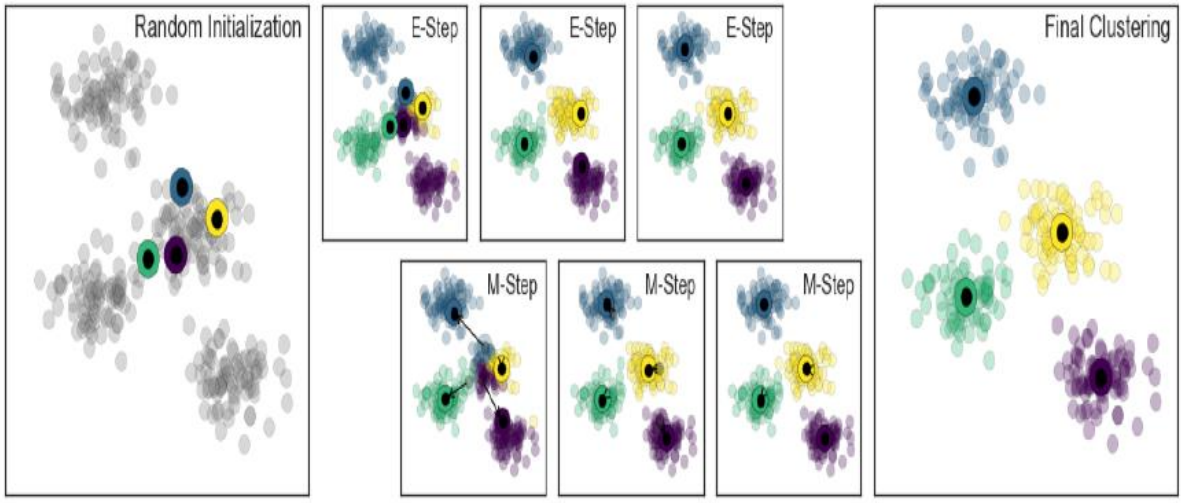
i.e.

$$\mu_j \leftarrow \frac{1}{m} \sum_{i=1}^{m} E[z_{ij}] x_i$$

E step: Calculate probabilities for unknown parameters for each instance

M step: Estimate parameters based on the probabilities

In k-means the probabilities are stored as instance weights.



Algorithm Hierarchical agglomerative

/* dissimilarity matrix $D(ij)$ is given */

1. Find minimal entry d_{ij} in D and merge clusters i and j
2. Update D by deleting column i and row j, and adding new row and column $i \cup j$
3. Revise entries using $d_{k,i\cup j} = d_{i\cup j,k} = \alpha_i d_{ki} + \alpha_j d_{kj} + \gamma |d_{ki} - d_{kj}|$
4. If there is more than one cluster then go to step 1.

The algorithm relies on a general updating formula. With different operations and coefficients, many different versions of the algorithm can be used to give variant clusterings.

Single linkage $d_{k,i\cup j} = \min(d_{ki}, d_{kj})$ and $\alpha_i = \alpha_j = \frac{1}{2}$ and $\gamma = -\frac{1}{2}$.
Complete linkage $d_{k,i\cup j} = \max(d_{ki}, d_{kj})$ and $\alpha_i = \alpha_j = \frac{1}{2}$ and $\gamma = \frac{1}{2}$.
Average linkage $d_{k,i\cup j} = \frac{n_i d_{ki}}{n_i+n_j} + \frac{n_j d_{kj}}{n_i+n_j}$ and $\alpha_i = \frac{n_i}{n_i+n_j}, \alpha_j = \frac{n_j}{n_i+n_j}$ and $\gamma = 0$.

Note: dissimilarity computed for every pair of points with one point in the first cluster and the other in the second.

$sim(C_i, C_j) = max(sim(p_x, p_y))$

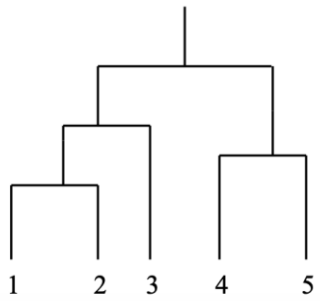
Single-Link Example

	P1	P2	P3	P4	P5
P1	1.00	0.90	0.10	0.65	0.20
P2	0.90	1.00	0.70	0.60	0.50
P3	0.10	0.70	1.00	0.40	0.30
P4	0.65	0.60	0.40	1.00	0.80
P5	0.20	0.50	0.30	0.80	1.00

	P1	P2	P3	P4	P5
P1	1.00	0.90	0.10	0.65	0.20
P2		1.00	0.70	0.60	0.50
P3			1.00	0.40	0.30
P4				1.00	0.80
P5					1.00

	12	P3	P4	P5
12	1.00	0.70	0.65	0.50
P3		1.00	0.40	0.30
P4			1.00	0.80
P5				1.00

	12	P3	45
12	1.00	0.70	0.65
P3		1.00	0.40
45			1.00



$sim(C_i, C_j) = min(sim(p_x, p_y))$

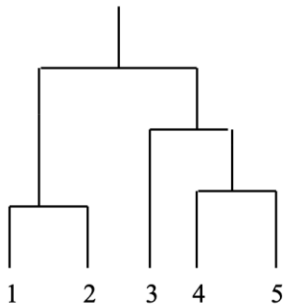
Complete-Link Example

	P1	P2	P3	P4	P5
P1	1.00	0.90	0.10	0.65	0.20
P2	0.90	1.00	0.70	0.60	0.50
P3	0.10	0.70	1.00	0.40	0.30
P4	0.65	0.60	0.40	1.00	0.80
P5	0.20	0.50	0.30	0.80	1.00

	P1	P2	P3	P4	P5
P1	1.00	0.90	0.10	0.65	0.20
P2		1.00	0.70	0.60	0.50
P3			1.00	0.40	0.30
P4				1.00	0.80
P5					1.00

	12	P3	P4	P5
12	1.00	0.10	0.60	0.20
P3		1.00	0.40	0.30
P4			1.00	0.80
P5				1.00

	12	P3	45
12	1.00	0.10	0.20
P3		1.00	0.30
45			1.00



Cluster quality visualisation - Silhouette plot

Key idea: compare each object's *separation* from other clusters relative to the *homogeneity* of its cluster.

For each object i, define its silhouette width $s(i) \in [-1, 1]$:

Let $a(i)$ be the average dissimilarity between i and elements of $P(i)$, i.e., cluster to which i belongs,

Let $d(i, C)$ be the average dissimilarity of i to elements of some *other* cluster C.

Let $b(i) = \min_C d(i, C)$. The *silhouette width* is

$$\frac{b(i) - a(i)}{\max(a(i), b(i))}$$