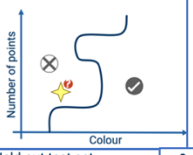
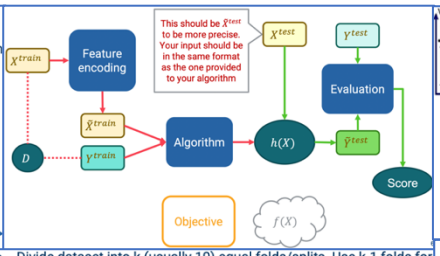
Lazy Learner
Stores the training examples and postpones generalising beyond these data until an explicit request is made at test time.

Eager Learner
Constructs a general, explicit description of the target function based on the provided training examples.





Occam's razor: More things should not be used than are necessary.

CASE 1: Plenty of data available -> Held-out test set
1) Train algorithm on **training set**
2) Tune hyperparameters on **validation set**
3) Estimate generalisation performance using the **test set**

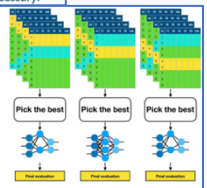
CASE 2: Limited data available -> Cross-validation
1) Separate dataset into **k folds**
2) Use 1 fold for testing and k-1 folds for **training+validation**
3) Repeat k times, using each fold as the **test set**
4) Estimate generalisation performance by **averaging results** across all the test folds

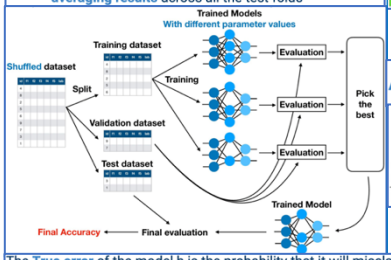
- Divide dataset into k (usually 10) equal folds/splits. Use k-1 folds for training+validation and one for testing.
- Iterate k times, each time testing on a different portion of the data.
- Performance on all k held-out test sets can then be averaged

Global error estimate $= \frac{1}{N}\sum_{i=1}^{N} e_i$ **CROSS-VAL**

Option 2: **NESTED CROSS-VAL**

- At each cross-validation step separate 1 fold for testing.
- Run an internal cross-validation over the remaining k-1 folds to find the optimal hyperparameters.
- Every fold will still have different hyperparameters, just chosen based on more data.





Eval metrics

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

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

Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

F1 score

$$F_\beta = (1+\beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

error rate = 1 – accuracy

N%	50%	68%	80%	90%	95%	98%	99%
Z_N	0.67	1.00	1.28	1.64	1.96	2.33	2.58

Macro-averaging: we are taking the average on the class level. We calculate metrics for each class and average them together.

$$P_{macro} = \frac{1}{4} \cdot (\frac{1}{3} + \frac{1}{1} + \frac{1}{2} + \frac{1}{2}) = 0.583$$

Micro-averaging: we are taking the average on the item level. For example, adding the TP, FP, TN, FN values for each class before calculating the metrics. **E.g. precision P**

$$P_{micro} = \frac{1+1+1+1}{(1+1+1+1)+(2+0+1+1)} = 0.5$$

Note: For binary and multi-class classification, micro-averaged P, R and F1 will be just equal to accuracy. This can vary for more complex measures and multi-label classification.

Macro-avg recall/precision/F: mean of R/P/F across classes

CM: True(row headers), Pred(col h.), count occur. If imbalanced data, down/upsample maj/min class

The **True error** of the model h is the probability that it will misclassify a randomly drawn example x from distribution D.

$$error_D(h) \equiv Pr[f(x) \neq h(x)]$$

The **Sample error** of the model h based on a data sample S:

$$error_S(h) \equiv \frac{1}{N}\sum_{x \in S} \delta(f(x), h(x))$$

n=number of samples
$\delta(f(x), h(x)) = 1$ **if** $f(x) \neq h(x)$
$\delta(f(x), h(x)) = 0$ **if** $f(x) = h(x)$

We want to know the true error but we can only measure the sample error.

Given a sample S with N >= 30, we can say with N% confidence the true error lies in the interval:

Scaling for the desired confidence level
Estimated standard deviation of the sample error
Both sides of the interval

$$error_S(h) \pm Z_N \sqrt{\frac{error_S(h) * (1 - error_S(h))}{n}}$$

Number of datapoints

Distinguish between verifying a hypothesis and exploring the data.

Benjamini & Hochberg (1995) offer an adaptive p-value:

1. Rank p-values from M experiments. $p_1 \leq p_2 \leq p_3 \leq ... \leq p_M$
2. Calculate the Benjamini-Hochberg critical value for each experiment. $z_i = 0.05\frac{i}{M}$
3. Significant results are the ones where the p-value is smaller than the critical value.

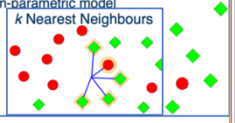
The **statistical tests** tell us if the means of the two sets are significantly different.
There are several statistical tests: Randomisation, T-test, Wilcoxon rank-sum, etc.
Randomisation test: randomly switch some predictions from both models and measure how often the new performance difference is greater or equal than the original difference.

Two-sample T-test: estimate the likelihood that two metrics (e.g. classification error) from different populations are actually different.

Paired T-test: estimating significance over multiple matched results, for example classification error over the same folds in cross-validation

Nearest Neighbour classifier

- Classify a test instance to the class label of the nearest training instance (according to some distance metric)
- Non-parametric model



If k>1, pick label w/ highest count among k neighbours. If distKNN, weight each neighbour before summing them

Manhattan distance (L^1-norm)
$d(x^{(i)}, x^{(q)}) = \sum_{k=1}^{K} |x_k^{(i)} - x_k^{(q)}|$

Euclidean distance (L^2-norm)
$d(x^{(i)}, x^{(q)}) = \sqrt{\sum_{k=1}^{K} (x_k^{(i)} - x_k^{(q)})^2}$

Chebyshev distance (L^∞-norm)
$d(x^{(i)}, x^{(q)}) = \max_{k=1}^{K} |x_k^{(i)} - x_k^{(q)}|$

Others: Mahalanobis distance, Hamming distance, ...

Algorithm: Decision Tree induction

1. Search for an 'optimal' splitting rule on training data
2. Split your dataset according to your chosen splitting rule
3. Repeat 1. and 2. on each new splitted subset

Information Gain

$$IG(dataset, subsets) = H(dataset) - \sum_{S \in subsets} \frac{|S|}{|dataset|} H(S)$$

Categorical vs. Real-valued attributes

Information Gain is the difference between the initial entropy and the (weighted) average entropy of the produced subsets.

$$IG(dataset, subsets) = H(dataset) - \sum_{S \in subsets} \frac{|S|}{|dataset|} H(S)$$

$$|dataset| = \sum_{S \in subsets} |S|$$

Entropy is defined as the ***average*** amount of information:

$$H(X) = -\sum_k^K P(x_k) log_2(P(x_k))$$

$$H(X) = -\int_x f(x) log_2(f(x))$$

Split rule 1: compute H(whole dataset D), IG(D, feature_1) ... IG(D, feature_n), pick feature w/ highest IG as 1st split. Split rule 2: do same as before, now D = each subset. For IG(D, feature_1), also need H(feature_1 = cat1) ... H(feature_1 = cat_n).
To construct tree (find splits) from text: identify the attributes and the categories for each, compute IG(D, attribute) for all attributes and pick the one w/ highest IG.
DON'T FORGET NORMALISING WEIGHTS IN IG + compute H(D) based on class labels

Definition of derivative:	$f'(a) = \lim_{h \to 0} \frac{f(a+h)-f(a)}{h}$	Constant multiple rule:	$\frac{d}{dx}[cf(x)] = cf'(x)$
Constant rule:	$\frac{d}{dx}(c) = 0$	Sum rule:	$\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$
Power rule:	$\frac{d}{dx}(x^n) = nx^{n-1}$	Difference rule:	$\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$
Exponential (e):	$\frac{d}{dx}e^x = e^x$	Product rule:	$\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Exponential (general):	$\frac{d}{dx}a^x = a^x \ln(a)$	Quotient rule:	$\frac{d}{dx}[\frac{f(x)}{g(x)}] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Natural log:	$\frac{d}{dx}(ln(x)) = \frac{1}{x}$	Chain rule:	$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$
General log:	$\frac{d}{dx}(log_a(x)) = \frac{1}{x ln(a)}$	Chain rule:	$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$

Our loss function (*cost function*) is the **sum-of-squares**:

$$E = \frac{1}{2}\sum_{i=1}^{N} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$$E = \frac{1}{2}\sum_{i=1}^{N} \left(ax^{(i)} + b - y^{(i)}\right)^2$$

MSE $\frac{1}{N}\sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2$

Binary classf (binary crossentropy / 1 output sigmoid neuron); multiclass classf (cat. crossentropy / >1 softmax neurons); regr (MSE, / one linear neuron); multilabel classf (1 binary crossentropy for each sigmoid output neuron / >1 sigmoid neurons)

LINEAR & LOGISTIC REGRESSION

Simple linear regression: 1 input feature

Multiple linear regression: many input features

e.g. 2 input feat. $\hat{y}^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b$

Param updates: use grad of SS loss wrt each param

The gradient is a **vector of all partial derivatives**

For a function $f : \mathbb{R}^K \to \mathbb{R}$, the gradient is

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_K} \end{bmatrix}$$

$$X = \begin{bmatrix} x^{(1)} & 1.0 \\ x^{(2)} & 1.0 \\ \vdots & \vdots \\ x^{(N)} & 1.0 \end{bmatrix} \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} \quad \theta = \begin{bmatrix} a \\ b \end{bmatrix}$$

Analytical sol.

$$\nabla_\theta E(\theta) = X^T(X\theta - y) = 0$$

$$\implies \theta^* = (X^T X)^{-1} X^T y$$

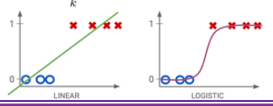
Great for directly finding the optimal parameter values. Not so great for large problems: matrix inversion has cubic complexity.

Numerical sol.: gradient descent (iterative) $\theta_i^{(t+1)} = \theta_i^{(t)} - \alpha \frac{\partial E}{\partial \theta_i^{(t)}}$

Logistic Regression Not actually regression

In logistic regression, we pass the regression output through a logistic function. This allows us to do binary classification (0 or 1).

$$\hat{y} = g(\sum_k \theta_k x_k) \qquad g(z) = \frac{1}{1+e^{-z}}$$



1H MLP:

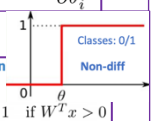
$$h = g_h(W_h^T x + b_h) \qquad h \in \mathbb{R}^{H \times 1} \quad W_h \in \mathbb{R}^{K \times H} \quad b_h \in \mathbb{R}^{H \times 1}$$

$$\hat{y} = g_{\hat{y}}(W_{\hat{y}}^T h + b_{\hat{y}}) \qquad \hat{y} \in \mathbb{R}^{C \times 1} \quad W_{\hat{y}} \in \mathbb{R}^{H \times C} \quad b_{\hat{y}} \in \mathbb{R}^{C \times 1}$$

$x \in \mathbb{R}^{K \times 1}$ most act. elem-wise except softmax

Perceptron: Bin. classf

It uses the **threshold function** function



$$h(x) = f(W^T x) = \begin{cases} 1 & \text{if } W^T x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron learning rule:

$$\theta_i \leftarrow \theta_i + \alpha(y - h(x))x_i$$

Can learn any linearly separable fn

Tanh activation: Similar shape to the sigmoid but ranges between -1 and 1. Input 0 also corresponds to output 0.

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

ReLU activation: Rectified linear unit. Linear in the positive part, but still non-linear overall.

$$f(x) = ReLU(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Softmax activation: Scales the inputs into a probability distribution. The largest input will be large, the rest will be small. All output values sum up to 1.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} log(\hat{y}_{i,c})$$

CE loss & grad w/ softmax

$$\frac{\partial L}{\partial z} = \frac{1}{N}(\hat{y} - y)$$

Maximising the logarithm is the same as maximising the original product, so let's take the log:

$$\sum_{i=1}^{N} y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)})$$

Turning this into a loss we can minimise gives us the **binary cross-entropy**:

$$L = -\frac{1}{N}\sum_{i=1}^{N} (y^{(i)} log(\hat{y}^{(i)}) + (1 - y^{(i)}) log(1 - \hat{y}^{(i)}))$$

This can be generalised to **categorical cross-entropy** for multi-class classification:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(i)} log(\hat{y}_c^{(i)})$$

Where C is the set of possible classes and $\hat{y}_c^{(i)}$ is the predicted probability of class c for datapoint i.

Classf tasks (objective): maximise the likelihood of network assigning correct labels to all inputs in our dataset

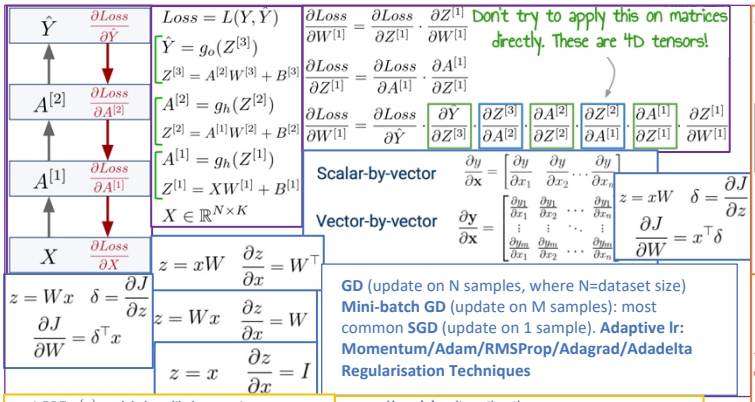
$$\prod_{i=1}^{N} p(y^{(i)} | x^{(i)}; \theta)$$

Tanh: $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ $g'(z) = 1 - g(z)^2$

ReLU: $g(z) = \begin{cases} z & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}$ $g'(z) = \begin{cases} 1 & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}$

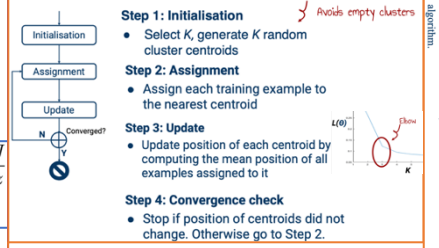
Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$ $g'(z) = g(z)(1 - g(z))$

Batching: combining the vectors of several datapoints into one matrix to improve speed and reduce noise



GD (update on N samples, where N=dataset size)
Mini-batch GD (update on M samples): most common **SGD** (update on 1 sample). **Adaptive lr: Momentum/Adam/RMSProp/Adagrad/Adadelta**
Regularisation Techniques

K-Means: Summary

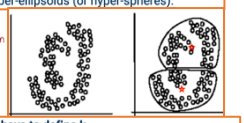


- **Simple**
  - Easy to understand and implement
- **Popular**
  - Used very often for clustering
- **Efficient**
  - Linear complexity
  - $O(TKN)$ for T iterations, K clusters, N examples. Since $K << N$ and $T << N$, k-means is considered a linear algorithm.

k-mode (cat. data): centroid = cluster mode. K-medoid: geometric median, not mean of locs (less sens. to outliers)

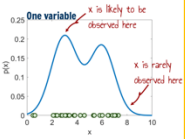
Step 1: Initialisation
Randomly select K training examples as cluster centroids
Step 2: Assignment
For each $i \in \{1, \ldots, N\}$ **Eucl dist**
$c^{(i)} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmin}} ||x^{(i)} - \mu_k||^2$
(To which cluster is training example $x^{(i)}$ assigned? $c^{(i)} \in \{1, \ldots, K\}$; Distance between training example (i) and cluster mean)
Step 3: Update
For each $k \in \{1, \ldots, K\}$
$\mu_k = \frac{\sum_{i=1}^{N} 1(c^{(i)} = k) \cdot x^{(i)}}{\sum_{i=1}^{N} 1(c^{(i)} = k)}$
(Mean of cluster k; Indicator function: 1(a) If a is True, return 1. Else return 0)
Step 4: Convergence check
If $\forall_k \left|\mu_k^t - \mu_k^{(t-1)}\right| < \epsilon$, stop. Else go to Step 2.
(Difference between updated mean and previous mean for each cluster is "insignificant enough")

k-means is not suitable for discovering clusters that are not hyper-ellipsoids (or hyper-spheres).



- **We have to define k**
  - K-means finds a local optimum
  - ... and is sensitive to the initial centroid positions
- Solutions:
  - Run multiple times, choose model with lowest cost $L(\Theta)$
  - Use better initialisation: e.g. K-means++

- A PDF $p(x)$ models how likely a continuous variable is to be observed within a particular **interval**
  - Note: No upper bound
  - $p(x) \geq 0$
  - $\int_a^b p(x)dx = 1$
  - Pfx $\in [a, b]$

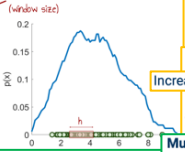


Density Estimation: estimate $p(x)$ from data

- Kernel density estimation
  - Compute $\hat{p}(x)$ by looking at training examples within a kernel function H (or Parzen window)

$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h^D} H\left(\frac{x - x^{(i)}}{h}\right)$
(Bandwidth (window size); Volume of window (D is number of dimensions); Kernel function)

$H(u) = \begin{cases} 1 & |u_j| < \frac{1}{2}; j = 1, \ldots, D \\ 0 & otherwise \end{cases}$
($X^{(i)}$ in/outside the window)



- Define kernel function H as a Gaussian: Smoother $\hat{p}(x)$

Increasing the bandwidth h smooths out $\hat{p}(x)$

$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{(2\pi h^2)^{\frac{D}{2}}} \exp\left(-\frac{||x - x^{(i)}||^2}{2h^2}\right)$

Gaussian fitting = minimising the negative log likelihood

- How do we quantify that we have fitted a good model?
  - Goal: model captures the probability of generating or observing data x within an interval
- **Likelihood**: measures the probability of observing data x from a dataset
  - $likelihood = p(\mathcal{X}|\theta) = \prod_{i=1}^{N} p(x^{(i)}|\theta)$
  - (θ: Parameters of your model (mean, covariance); iid. assumption)
- **Negative Log-Likelihood** is often used:
  - $\mathcal{L} = -\log p(\mathcal{X}|\theta) = -\sum_{i=1}^{N} \log p(x^{(i)}|\theta)$
  - (Negative: Turn it to a minimization problem; LOG: for numerical stability (multiplication of small probabilities -> bad))

Mixture Models

$p(x) = \sum_{k=1}^{K} \pi_k p_k(x)$
(π_k: Mixing proportions)

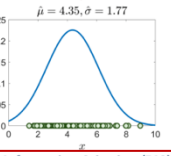
$0 \leq \pi_k \leq 1$
$\sum_{k=1}^{K} \pi_k = 1$

Univariate Gaussian (Normal) distribution

$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
(Larger variance: flatter and more spread out; Small variance: more peaked and tighter)

Compute $\hat{p}(x)$ by fitting the parameters $\hat{\mu}$ and $\hat{\sigma}^2$ to the training examples

$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x^{(i)}$
$\hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^{N} (x^{(i)} - \hat{\mu})^2$
$\hat{p}(x) = \mathcal{N}(x|\hat{\mu}, \hat{\sigma}^2)$

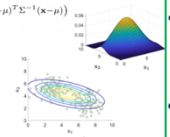


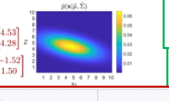
Multivariate Gaussian (Normal) distribution

$\mathcal{N}(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D |\Sigma|}} \cdot \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu)\right)$

Compute $\hat{p}(\mathbf{x})$ by fitting the parameters $\hat{\mu}$ and $\hat{\Sigma}$ to the training examples

$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}^{(i)}$
$\hat{\Sigma} = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{x}^{(i)} - \hat{\mu})(\mathbf{x}^{(i)} - \hat{\mu})^T$
$\hat{\mathbf{p}}(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\hat{\mu}, \hat{\Sigma})$

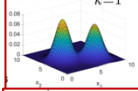




Gaussian Mixture Models (most popular)

$p(\mathbf{x}|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$

$0 \leq \pi_k \leq 1$
$\sum_{k=1}^{K} \pi_k = 1$
$\theta = \{\pi_k, \mu_k, \Sigma_k : k = 1, \ldots, K\}$



- Minimize the **Bayesian Information Criterion (BIC)**

$BIC_K = \mathcal{L}(K) + \frac{P_K}{2} log(N)$
(Number of parameters; Number of examples; Negative Log Likelihood (encourages fitting of data); Penalises complex models)

Number of parameters for 2D Gaussians: $P_K = 6*K - 1$ (2 for mean, 3 for covariance, 1 for mixing proportion)

$N_k = \sum_{i=1}^{N} r_{ik}$

GMM-EM

Step 1: Initialisation
- Select K, randomly initialise all parameters

Step 2: E-Step
- Compute the responsibilities

$r_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(i)}|\mu_k, \Sigma_k)}{\sum_j^K (\pi_j \mathcal{N}(\mathbf{x}^{(i)}|\mu_j, \Sigma_j))}$

Step 3: M-Step
- Update the mean
- Update the covariance
- Update the mixing proportions

$\hat{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N} (r_{ik} \mathbf{x}^{(i)})$
$\mathbf{\Sigma}_k = \frac{1}{N_k} \sum_{i=1}^{N} (r_{ik} (\mathbf{x}^{(i)} - \hat{\mu}_k)(\mathbf{x}^{(i)} - \hat{\mu}_k)^T)$
$\pi_k = \frac{N_k}{N}$

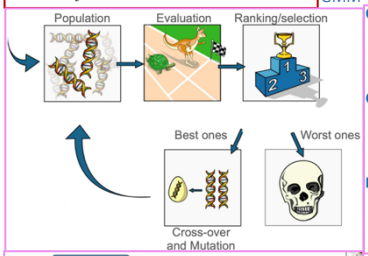
Step 4: Convergence check
- Stop if converged (parameters not changed, or likelihood stagnated). Otherwise go to Step 2.

$\sigma_k^2 = \frac{1}{\sum_{j=1}^{N} r_{jk}} \sum_{i=1}^{N} (r_{ik} (x^{(i)} - \mu_k)^2)$

GMM = simplicity of GMs + expressivity of KDE

K-means	GMM-EM
Objective function: Minimize average mean squared distance	Objective function: Maximize log-likelihood
EM-like algorithm - Assignment: Assign points to cluster - Update: Optimise cluster	EM algorithm - E-step Compute posterior probability of membership - M-step: Optimise parameters
Performs hard assignment during Assignment step	Performs soft assignment during E-step
Assume spherical clusters with equal probability for each cluster	Can be used for non-spherical clusters Can generate clusters with different probabilities

Define GA/EA: genotype and phenotype (how go from g to p); fitness fn; selection operator; cross-over op; mutation op; hyperparams like pop size



Genetic algorithm: **Optim param list of ints**
- Genotype: binary string of fixed size (ex. 01001010)
- Cross-over: swap portions of the string
- mutation: Random flip of bits

Genetic programming:
- Genotype: Program represented as tree (often in LISP)
- Cross-over: Swap portions of trees
- mutation: Change the symbols in the tree

Evolutionary Strategies: **Optim real-val params**
- Genotype: String of floats/doubles
- Cross-over: usually not used
- Mutation: Draw from a Gaussian distribution

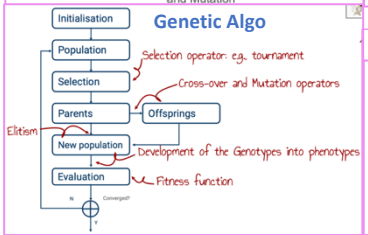
Fitness fn: represent prblm to solve (max. this fn). **Gen/Phen**: potential sols. **Selection op**: select parents for next gen (use: Biased Roulette Wheel; Tournament; Elitism). **Crossover op**: how we combine parents' traits to produce offspring. **Mutation op**: random var to explore nearby sols.

Process: **Biased roulette wheel**
- Compute the probability p_i to select an individual from the population:
- Compute the cumulative probability
- Randomly generate (uniform distribution) a number r between 0 and 1.
- Select the individual x_i so that:

$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \quad q_i = \sum_{j=1}^{i} p_j \quad q_{i-1} < r \leq q_i$

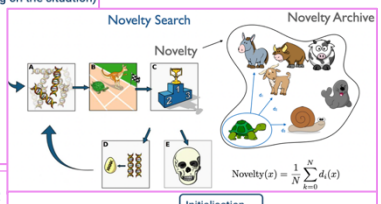
Tournament: Sampling w/ replacement (parents can be selected >1, until we have selected enough parents)

Simple version: **Tournament**
1. Randomly draw two individuals from the population
2. Select the best out of the two
3. Repeat this until you have enough parents.

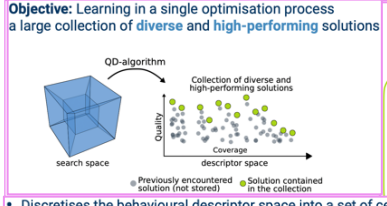


E.g. ES: **(mu+lbmda)-ES**
1. Randomly generate a population of $(\mu + \lambda)$individuals.
2. Evaluate the population — Pick st. lambda/mu = 5
3. Select the μ best individuals from the population as parents (called x)
4. Generate λ offsprings (called y) from the parents:
   $y_i = x_j + \mathcal{N}(0, \sigma) \qquad j = randi(\mu)$
5. Population = union of the parents and offspring:
   $\mathbf{pop} = (\cup_i^{\lambda} y_i) \cup (\cup_j^{\mu} x_j)$ **Next gen=mu+lbda**
6. Return to 2

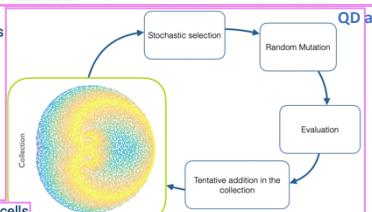
Sigma: mut. rate of mutation op
- Adapting sigma over time (or depending on the situation)
- Concept: Add sigma in the genotype:
  - $x_i' = \{x_i, \sigma_i\}$
  - $\sigma_i = \sigma_i \exp(\tau_0 \mathcal{N}(0,1))$
  - $y_i = x_i + \sigma_i \mathcal{N}(0,1)$
- τ_0 Is the learning rate
- heuristic: $\tau_0 \propto 1/\sqrt{(n)}$ With n = dimension of the genotype.

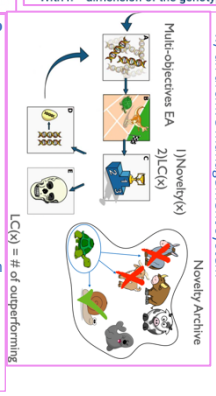


Objective: Learning in a single optimisation process a large collection of **diverse** and **high-performing** solutions

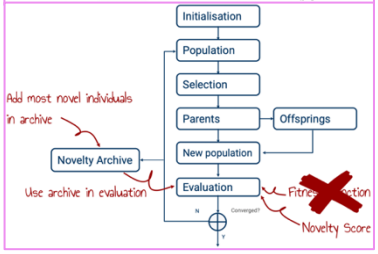


QD algo





NS w/ local comp: QD algo when used w/ an archive management system



- Discretises the behavioural descriptor space into a set of cells
- **Addition mechanism:**
  - Each new solution goes to the cell corresponding to its BD.
  - If the cell is empty the new solution is added to the grid
  - If the cell is already occupied, the solution with the best fitness is kept
- Hyper-parameter: size of the cells (or resolution of the grid)
- Advantage: Easy to implement
- Drawback: Density not necessarily uniform

E.g. QD algo: MAP-Elites = Grid + uni rand select. **Coverage** = num unique sol / total num sol (or num of cells filled w/ 1 sol in the grid) **Diversity** = archive size. **Performance** = max or mean fitness score in archive. **Convergence speed** = btw diversity and performance metrics. **QD-score** = sum of fitness scores of all sol in archive (>0).

Rappel: EuclDist(p,q) = sqrt((q_x – p_x)**2 + (q_y – p_y)**2), where p and q are R2 vectors. Most novel sol = sol whose BD is furthest away from its neighbours' BD.