1 Definitions

- **Artificial Intelligence**: computers to mimic human behavior and intelligence.
- **Machine Learning**: subset of AI, using statistical methods to improve with experience.
- **Deep Learning**: subset of ML, using multi-layered neural networks to model complex patterns in data.
- **Supervised Learning**: learn a function that maps inputs to output labels based on example input-output pairs.
- **Unsupervised Learning**: learn patterns in input data without labeled outputs (clustering, dimensionality reduction).
- **Reinforcement Learning**: learn a policy to maximize cumulative reward through trial and error in an environment.
- **Classification**: predict a discrete label from a fixed set of classes (*e.g. spam detection*).
- **Regression**: predict a continuous value.
- **Lazy Learner**: stores data & make preds based on similarity to training set (*e.g. k-NN*).
- **Eager Learner**: build model from data & make preds using model (*e.g. decision trees*).
- **Non Parametric Model**: complexity grows with data (*e.g. k-NN*).
- **Parametric Model**: fixed number of params.
- **Linear Model**: data linearly separable.
- **Non Linear Model**: to make *linear*, perform feature space transformation *(Kernel trick in SVMs & non-linear activation funcs in NNs)*.
- **Underfitting**: model too simple to capture underlying patterns: ***HIGH BIAS, LOW VARIANCE***.
- **Overfitting**: model too complex & captures noise: ***HIGH VARIANCE, LOW BIAS***.
- **Instance Based Learner**: lazy learner where model stores training set, making preds based on similarity. *Model only built when pred required.*

Dataset X is split into *train* and *test* sets. Each **feature** $x_k^{(i)}$ is **standardised** as $\tilde{x}_k^{(i)} = \frac{x_k^i - \mu_k}{\sigma_k}$. Too many features $\mapsto$ **curse of dimensionality**: data too sparse, overfitting occurs.

2 K-NNs

A **K-NN** classifier assigns label based on the most popular label amongst K nearest neighbours. *K is odd.* Increasing K:

- *Smoother decision boundary* (**higher bias**)
- *Less sensitive to training data* (**lower variance**)

We also need a *distance metric*:

- **Manhattan(ℓ_1)**: $d(x_i, x_q) = \sum_{j=1}^{n} |x_{ij} - x_{qj}|$
- **Euclidian(ℓ_2)**: $d(x_i, x_q) = \sqrt{\sum_{j=1}^{n} (x_{ij} - x_{qj})^2}$
- **Chebyshev(ℓ_∞)**: $d(x_i, x_q) = \max_{j=1}^{n} |x_{ij} - x_{qj}|$

A **distance weighted K-NN** weights its neighbours by their dist. To find weights:

- **Inverse**: $w_i = \frac{1}{d(x_i, x_q)}$.
- **Gaussian**: $w_i = \frac{1}{\sqrt{2\pi}} \exp(-\frac{d(x_i, x_q)^2}{2})$.

Now incr K has less effect on classification (good). When $K = N$, this is a **global method**. Otherwise, its a **local method**. DKNNs are more robust to noisy data, but suffer from curse of dim. K-NNs will also not filter out irrelevant features. KNN does regression by computing *mean* across K NNs.

3 Decision Trees

An *eager learner* algorithm that:

1. Search for an optimal splitting rule.
2. Split the dataset according to the rule.
3. Repeat on each new subset.

Entropy: measure of uncertainty of a RV, the expected amount of **information** required to fully define a random state. *Low entropy variables are predictable, high entropy vars are not.* **Information** $I(x) = \log_2(K)$ when x takes K states, $K = \frac{1}{P(x)}$. So, $I(x) = -\log_2(P(x))$. The **avg info** is:

$$H(x) = -\sum_{k}^{K} P(x_k)\log_2(P(x_k))$$

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

For each rule, the **information gain** is $\mathrm{IG}(D,S) = H(D) - \sum_{s\in S} \frac{|s|}{|D|} H(s)$ where D is the *dataset*, S is the *subset* & $|D| = \sum_{s\in S} |s|$. Split *ordered* vals by **threshold** and *categorical* vals by **symbol**. To stop *overfitting*:

- **Stop Early**: set *max depth* for decision tree.
- **Prune**: Loop through connected to leaf nodes, turn into a leaf with majority class label. Eval pruned tree on validation set, prune if accuracy higher than unpruned. Repeat until all nodes tested.

Many decision trees make a **random forest**. *Regression* done by leaf nodes predicting real number.

4 Evaluation

Data split into *shuffled (unordered)* **training/test**. To tune **hyperparams**, also split a **validation** set to eval hyperparams. After hyperparameter tuning, retrain model on combined *train/valid* sets to get best model. Then evaluate on test set.

4.1 Cross Validation

When data is limited, 3 sets is wasteful. Instead divide into k **folds** with $k-1$ folds for *train/valid* and 1 for test. Repeat k times with different test folds. Final performance averaged across k runs.

$$\text{Global Error Estimate} = \frac{1}{k}\sum_{i=1}^{k} e_i$$

Where e_i is the error on fold i. For hyperparameter tuning, we can either:

- 1 test fold, 1 valid fold, $k-2$ train folds. *Finds optimal hyperparameters per fold.*
- 1 test fold, $k-1$ cross valid folds. *Expensive & each fold has its own hyperparameters.*

4.2 Evaluation Metrics

Conf Mat	**Pred Pos**	**Pred Neg**
True Pos	TP	FN
True Neg	FP	TN

$$\Longrightarrow \begin{pmatrix} \mathrm{TP} & \mathrm{FN} \\ \mathrm{FP} & \mathrm{TN} \end{pmatrix}$$

- **Accuracy**= $\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$ (*proportion correct*).
- **Classification Error**= $1 -$ Accuracy.
- **Precision**= $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$ (*proportion pos correct*).
- **Recall**= $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$ (*prop actual pos correct*).
- **Macro Avg precision/recall** calc per class, then average them; *treating classes equally*.
- **Micro Avg precision/recall** sum TP, FP, FN across all classes, then calc; *treating all examples equally*.
- **F Score** *combines precision and recall*: $F_\beta = (1+\beta^2)\frac{\text{precision}\times\text{recall}}{\beta^2\times\text{precision}+\text{recall}}$ where $\beta > 0$.
- **Mean Sq Err**= $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$.
- **RMSE**= $\sqrt{\mathrm{MSE}}$ *same units as target var*.

If data distribution *imbalanced*, we should **normalise** confusion matrix rows; or **upsample/-downsample** data to balance classes.

4.3 Statistical Significance

A model **true error** is the prob it misclassifies a random sample, $err_D(h) = P(f(x) \neq h(x))$. The **sample error** is based on data sample S: $err_S(h) = \frac{1}{N}\sum_{x\in S}\delta(f(x), h(x))$ where $\delta(a,b) = \begin{cases} 1 & a \neq b \\ 0 & a = b \end{cases}$. Given a sample S with $N \geq 30$, we can estimate $\mathrm{err}_D(h)$ with an α% confidence interval:

$$\mathrm{err}_S(h) \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\mathrm{err}_S(h)(1-\mathrm{err}_S(h))}{N}}$$

Statistical tests say if means of two sets are significantly different:

- **Randomisation**: Randomly switch preds between two models, calc diff in acc. Repeat to get distr of diffs.
- **Two Sample T**: Estimate likelihood that two metrics from diff populations are diff.
- **Paired T**: Estimate significance over many matched results.

P-hacking is the misuse of data to find patterns that appear significant.

5 Linear Regressions

Linear regression: a dataset $\{\langle x^{(1)}, y^{(1)}\rangle, \cdots, \langle x^{(N)}, y^{(N)}\rangle\}$ consisting of inputs $x^{(i)}$ and outputs $y^{(i)}$ is used to learn a $f : X \to Y$ such that $\forall i \in \{1, \cdots, N\}. f(x^{(i)}) = y^{(i)}$. Assuming that f is linear, train by minimising **loss func** between pred outputs and true outputs. Sum of squares **loss func**:

$$E = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2 \quad \text{where } \hat{y}^{(i)} = f(x^{(i)})$$

Good loss funcs are easily differentiable. To minimise, use **gradient descent**. To do this, update params with their partial derivatives:

$$\frac{\partial E}{\partial a} = \frac{\partial}{\partial a}\frac{1}{2}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2 = \sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})x^{(i)}$$

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

5.1 Gradient Descent (LR)

Gradient descent updates params by taking small steps in the neg dir of the partial derivatives:

```python
for epoch in range(num_epochs):
  y_pred = a * X + b
  a = a - lr * sum((y_pred - Y) * X)
  b = b - lr * sum(y_pred - Y)
  rmse = sqrt(mean(square(y_pred - Y)))
  print(f"{epoch+1}: {a}, {b}, {rmse}")
```

Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ is the gradient of its partial derivs: $\nabla_\theta f(\theta) = \left[\frac{\partial f(\theta)}{\partial\theta_1} \quad \cdots \quad \frac{\partial f(\theta)}{\partial\theta_n}\right]^T$.

6 Neural Networks

A **neuron** has inputs $x_1, \cdots, x_m$ & weights $\theta_1, \cdots, \theta_m$ & bias b, producing output $\hat{y}$. It also has an **activation func** g that introduces non-linearity: $\hat{y} = g\left(\sum_{i=1}^{m}\theta_i x_i + b\right)$. In notation, omit bias by adding extra input $x_0 = 1$ with weight $\theta_0 = b$. We can rewrite this with vector notation using $W \in \mathbb{R}^{m\times 1}$ and $x \in \mathbb{R}^{m\times 1}$: $\hat{y} = g(W^T x)$. Neurons are connected in *parallel*, so each neuron detects something different. By connecting them *serially* we learn higher order feats.
By connecting $x \to h_1 \to h_2 \to \hat{y}$, we have:

- $h_1 = g_{h_1}(W_{h_1}^T x + b_{h_1})$
- $h_2 = g_{h_2}(W_{h_2}^T h_1 + b_{h_2})$
- $\hat{y} = g_{\hat{y}}(W_{\hat{y}}^T h_2 + b_{\hat{y}})$

6.1 Perceptron

Perceptrons dont use grad desc. They use a **threshold func** as the activation func: $g(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$. The learning rule: $\theta_i \leftarrow \theta_i + \alpha(y - h(x))x_i$. Then:

- If desired output y matches the pred $h(x)$, no update is made.
- If $y = 1$ & $h(x) = 0$, weights *increased* to make $h(x)$ more likely to be 1.
- If $y = 0$ & $h(x) = 1$, weights *decreased* to make $h(x)$ more likely to be 0.

With this, we learn any **linearly separable func**. The activation func is sharp and non-differentiable, so cannot be used with gradient descent.

6.2 Activation Functions

- **Linear** $g(z) = z$ for *linearly seperable* data. Reduces multi-layer net to single, not desirable. $g'(z) = 11$
- **Sigmoid** $g(z) = \frac{1}{1+e^{-z}}$ maps $z \mapsto (0,1)$, good for *binary classification*. $g'(z) = g(z)(1 - g(z))$
- **Tanh** $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ maps $z \mapsto (-1,1)$, good for *binary classification*. $g'(z) = 1 - g(z)^2$
- **ReLU** $g(z) = \max(0, z)$ maps $z \mapsto [0, \infty)$. Efficient & mitigates gradient vanishing. $g'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
- **Softmax** $g(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ scales values into a *probability distribution*. $\frac{\partial L}{\partial z} = \frac{1}{N}(\hat{y} - y)$

6.3 Loss Functions

E, Optimised in grad desc: $\theta_i \leftarrow \theta_i - \alpha\frac{\partial E}{\partial\theta_i}$

$$\frac{\partial \mathrm{MSE}}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i)$$

Cross entropy loss is $\prod_{i=1}^{N} p(y_i \mid x_i; \theta)$. Its **log likelihood** for *binary* data is:

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

For *categorical*, where C is set of possible categories: $L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$.

6.4 Backpropagation

Backpropogation optmises grad desc for multi-layer nets, avoiding recalcing the partial derivatives of each layer. A **forward pass** computes the outputs of each layer, and a **backward pass** computes the gradients of each layer using the chain rule. For example:

1. Receive the gradient from the next layer:
   $$\frac{\partial E}{\partial Z} \in \mathbb{R}^{N\times k},$$
   where $Z = XW + b$ is the matrix of pre-activation values. N = batch size, k = number of neurons.
2. To update parameters, compute gradients w.r.t. W and b. Because $Z = XW + b$, the derivative of Z w.r.t. W is X, so
   $$\frac{\partial E}{\partial W} = X^T\frac{\partial E}{\partial Z}.$$
   Each bias affects all samples equally, so
   $$\frac{\partial E}{\partial b} = \sum_{i=1}^{N}\frac{\partial E}{\partial z_i}.$$
3. To pass gradients to the previous layer, compute
   $$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Z}W^T,$$
   since changes in X affect Z through multiplication by W.
4. For the activation function $A = g(Z)$, apply the chain rule:
   $$\frac{\partial E}{\partial Z} = \frac{\partial E}{\partial A} \circ g'(Z),$$
   where $\circ$ denotes elementwise multiplication.
5. The quantity $\frac{\partial E}{\partial A}$ is the gradient *received from the next layer*, because A is that layer's input. For example, if the next layer is linear with $Z_{\text{next}} = AW_{\text{next}} + b_{\text{next}}$, then
   $$\frac{\partial E}{\partial A} = \frac{\partial E}{\partial Z_{\text{next}}}W_{\text{next}}^T.$$

6.5 Gradient Descent (NN)

Gradient descent iteratively trains a model. With **learning rate** α, update weights $W' \leftarrow W - \alpha\frac{\partial E}{\partial W}$. In **batched** gradient descent:

1. Initialise weights W randomly.
2. Until convergence, loop over **batches**, compute grad of batch only, update weights.

Loss surfaces are complex and we want to avoid local minima. LR too low $\to$ wont converge, too high $\to$ overshoot minima:

- **Adaptive LR** has diff LR per parameter, taking bigger steps if the gradient is small, and vice versa.
- **LR decay** takes smaller steps the closer to the minimum: $a' \leftarrow ad$, $d \in (0,1)$.

6.6 Weight Initialisation

- **Zero**: all neurons learn same features.
- **Normal**: draw weights from $\mathcal{N}(0,\sigma^2)$.
- **Xavier Gorot**: $W \sim U\left(\pm\sqrt{\frac{6}{n_{in}+n_{out}}}\right)$ where n is the num of inputs & outputs, keeps the variance of activations and backpropagated gradients roughly the same across layers.

6.7 Data Normalisation

Helps with convergence, as weight updates $\propto$ input data. Methods include:

- **Minmax**: $x' = a + \frac{(x-\min(x))(b-a)}{\max(x)-\min(x)}$ scales data to $[a,b]$.
- **Standardisation**: $x' = \frac{x-\mu}{\sigma}$ where μ, σ are the mean, variance of input data. Gives data with mean 0 and var 1.

Scaling values must only be calculated on the training set.

6.8 Gradient Checking

Verifies backprop is correctly computing:

- **Weight difference**: $w_t = w_{t-1} - \alpha\frac{\partial E}{\partial w_{t-1}}$.
- Perturb weight and check **loss difference**: $\frac{\partial E}{\partial w} = \lim_{\epsilon\to 0}\frac{E(w+\epsilon)-E(w-\epsilon)}{2\epsilon}$.

Both methods should give very similar values of $\frac{\partial E}{\partial w}$.

6.9 Overfitting

To prevent overfitting, *(1)* decrease **capacity**, *(2)* use more **training data**, *(3)* **stop early** by using **validation set** to monitor perf improvement over epochs, *(4)* **dropout** by randomly disabling neurons during training preventing **coadaptation**, *(5)* **regularisation**, add info or constrants to prevent overfitting:

- **L2** add square weights to loss func, *encouraging sharing between features*: $J(\theta) = E(y,\hat{y}) + \lambda\sum_w w^2$. So $w \leftarrow w - \alpha\left(\frac{\partial E}{\partial w} + 2\lambda w\right)$.
- **L1** add absolute weights to loss func, *encouraging sparsity*: $J(\theta) = E(y,\hat{y}) + \lambda\sum_w |w|$. So, $w \leftarrow w - \alpha\left(\frac{\partial E}{\partial w} + \lambda\mathrm{sign}(w)\right)$.

7 Clustering

Cluster: set of instances similar to each other but dissimilar to instances in other clusters. **Clustering** is grouping instances in some feature space into clusters. **K-means** clustering:

1. Randomly select K initial cluster centroids. *Randomly select $\mu_0,\cdots,\mu_k$.*
2. Assign each data point to nearest centroid. $\forall i \in \{1,\dots,n\}.\left[c^{(i)} := \arg\min_j \|x^{(i)} - \mu_j\|^2\right]$
3. Recalculate centroids as mean of its points. $\forall j \in \{1,\dots,k\}.\left[\mu_j := \frac{\sum_{i=1}^n 1\{c^{(i)}=j\}x^{(i)}}{\sum_{i=1}^n 1\{c^{(i)}=j\}}\right]$ *where*
   $$1\{\cdot\} = \begin{cases} 1 & \textit{if condition is true} \\ 0 & \textit{otherwise}\end{cases}$$
4. If not converged, restart from *(2)*.

To pick K, use *cross validation* or **elbow method**:

1. Run K-means multiple times with diff K.
2. Keep track of cost $L(\Theta)$ for each K.
3. Plot $L(\Theta)$ against K & look for *elbow point* where the decrease in cost starts slows down. This point is a good choice for K.

K-means is *simple* & *efficient*, but K is pre-specified, finds a *local optimum*, needs *dist func*, sensitive to *outliers* and does not handle *hyper-ellipsoidal clusters*.

8 Probability Density Estimation

Can be **non-parametric** (*low bias, high var*) or **parametric** (*high bias, low var*) (assuming data distribution).

8.1 Histograms

1. Divide data range into k equal-width bins.
2. Count the number of data points in each bin.
3. Estimate PDF as normalized counts per bin.
4. Choice of bin width affects estimate: too wide loses detail, too narrow adds noise.

8.2 Kernel Density Estimation

Computes $\hat{p}(x)$ by looking at training examples in a kernel function H:

$$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^n N\frac{1}{h^D}H\left(\frac{x-x^{(i)}}{h}\right)$$

Where N is num of training examples, h is **bandwidth** (*window size*) & D is num of dimensions. A simple kernel function is the **uniform kernel**:

$$H(u) = \begin{cases} 1 & \text{if } \forall j \in \{1,\cdots,D\}.|u_j| \le \frac{1}{2} \\ 0 & \text{otherwise}\end{cases}$$

Another common kernel is **Gaussian**:

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

Increasing h smooths estimate, decreasing h adds noise sensitivity.

8.3 Parametrics Methods

Assume data has **uniform Gaussian dist**:

$$\mathcal{N}(x \mid \mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Then $\hat{p}(x)$ found by fitting μ, σ^2:

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

The **Multivariate Gaussian dist** generalizes univariate case to many dimensions:

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

Then $\hat{p}(x)$ found by fitting μ, σ^2:

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

A **Mixture Model** improves *bias-var* tradeoff, combining many distributions:

$$p(x) = \sum_{k=1}^K \pi_k p_k(x \mid \theta_k)$$
$$\text{where } 0 \leqq \pi_k \le 1 \wedge \sum_{k=1}^K \pi_k = 1$$

A **Gaussian Mixture Model (GMM)**:

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

8.4 Likelihood

Quantifies how well model fits data as the *probability of observing x* from a dataset:

$$p(\mathcal{X} \mid \theta) = \prod_{i=1}^N p(x^{(i)} \mid \theta)$$

where θ are model params. *Negative log-likelihood* makes this a minimisation problem:

$$\mathcal{L} = -\log p(\mathcal{X} \mid \theta) = -\sum_{i=1}^N \log p(x^{(i)} \mid \theta)$$

When Gaussian fitting, we are actually minimising log likelihood! This can be proven by setting $\frac{\partial\mathcal{L}}{\partial\mu} = 0$ and $\frac{\partial\mathcal{L}}{\partial\sigma^2} = 0$ to find the minima.

9 GMM-EM Algorithm

GMMs model complex dists, but have lots of params to optimise: $\theta = \{\pi_k,\mu_k,\Sigma_k \mid k = 1,\dots,K\}$. **GMM-EM** fits a GMM to data using the **Expectation-Maximization (EM)** algorithm:

1. Choose K for the number of components, randomly initialise params $\theta^{(0)}$.
2. **Expectation**: Compute **responsibilities** per data $x^{(i)}$ and component k:
   $$r_{ik} = \frac{\pi_k\mathcal{N}(x^{(i)} \mid \mu_k,\Sigma_k)}{\sum_{j=1}^K \pi_j\mathcal{N}(x^{(i)} \mid \mu_j,\Sigma_j)}$$
3. **Maximization**: Per component: update the *means* $\hat{\mu}_k$, *covariances* $\hat{\Sigma}_k$, and *mixture proportions* $\hat{\pi}_k$:
   $$\hat{\mu}_k = \frac{1}{N_k}\sum_{i=1}^N r_{ik}x^{(i)}$$
   $$\Sigma_k = \frac{1}{N_k}\sum_{i=1}^N r_{ik}(x^{(i)} - \hat{\mu}_k)(x^{(i)} - \hat{\mu}_k)^T$$
   $$\pi_k = \frac{N_k}{N} \quad \text{where } N_k = \sum_{i=1}^N r_{ik}$$
4. If not **converged**, repeat from *(2)*.

Like K-means, GMM-EM converges to local optima. To find K we minimise the **Bayesian Information Criterion (BIC)**, which takes into account the negative log-likelihood and model complexity (*Occam's razor*): $BIC_K = \mathcal{L}(K) + \frac{P_K}{2}\log N$ where $P_K = 6K - 1$ is the number of parameters for 2D gaussians. *Or, we could use cross validation:*

1. Split into training and validation sets.
2. Fit GMM-EM on training set for different K.
3. Evaluate log-likelihood on validation set.

9.1 VS K-Means

In both cases:

1. We specify K clusters/components.
2. Convergence happens when assignments/-params stabilise.
3. Sensitive to initialisation.

Often K-means is run first to initialise GMM-EM.

GMM-EM does **soft clustering**, encoding to what degree each data point belongs to each cluster, while K-means does **hard clustering** - assigns each data point to exactly one cluster.

GMM-EM can also generate clusters with different probabilities and non-spherical clusters.

10 Evolutionary Algorithms

An optimisation algorithm for **black box funcs** (*unknown grad*), a *reinforcement learning problem*. The concept is:

1. Maintain **population** (*different genotypes*).
2. Eval **fitness** on black box func (*phenotype*).
3. Fittest individuals will **reproduce**.

There are three main operators:

- **Selection**: *who reproduces?*
- **Crossover**: Mixes parent genotypes.
- **Mutation**: Type and frequency variation applied to genotypes.

Easy to *parallelize* but slower than grad desc when gradient known and problem simple.

10.1 Genetic Algorithms

- **Fitness func** is the problem to solve. *Maximising* should lead to optimal solution.
- **Genotype & Phenotype** represent potential solutions. *Fed into FF (e.g. a binary string)*.
- **Stopping Critereon** usually specific *fitness val*, *generation limit*, or *convergence*.

In **Biased Roulette Wheel Selection**, $p_i = \frac{f_i}{\sum_{j=1}^N f_j}$ where f_i is the fitness of individual i. For random $r \in [0,1]$, the individual where the cum prob exceeds r: $q_i = \sum_{j=1}^i p_j \ge r$. In **tournament selection**:

1. Randomly draw 2 individs from population.
2. Select the one with higher fitness as a parent
3. Repeat until enough parents are selected.

This method is less susceptible to premature convergence, and easier to parallelise.

Elitism ensures best individs selected without alteration, preventing losing best solutions to random chance. Usually $\approx$ 10% of the population.

10.2 Evolutionary Strategies

Here, the *genotype* is a **list of reals**, parent *selection* **uniform** & *mutation* generated from a **gaussian**. In **$\mu + \lambda$ ES**:

1. Randomly generate $\mu + \lambda$ individuals.
2. Evaluate population.
3. Select μ best individuals as parents.
4. Generate λ offsprings y where $y_i = x_j + \mathcal{N}(0,\sigma)$ & x_j is a randomly selected parent.
5. The new population is $(\bigcup_{i=1}^{\mu} x_i) \cup (\bigcup_{i=1}^{\lambda} y_i)$.
6. Repeat from *(2)*.

Usually $\frac{\lambda}{\mu} \approx 5$. We must choose σ st:

- Large σ converges quickly, hard to refine.
- Small σ refines well, but longer convergence & can get stuck in local optima.
- We could update σ over time by adding σ to the genotype:
  $$x'_j = \{x_j,\sigma_j\}$$
  $$\sigma_i = \sigma_j\exp(\tau_0\,\mathcal{N}(0,1))$$
  $$y_i = x_j + \sigma_i\,\mathcal{N}(0,1)$$
  Where τ_0 is the learning rate. Heuristically, $\tau_0 \propto \frac{1}{\sqrt{d}}$ where $d = \dim(x)$.

10.3 Novelty Search

Uses **archive** to store prev seen behaviours. **Novelty** is avg dist to k nearest neighbours in archive $Nov(x) = \frac{1}{N}\sum_{i=1}^N d(x,x_i)$. For the dist metric, define a space of **behavioral descriptors**, which capture important aspects of the phenotype. Instead of optimising solution *quality*, NS optimises *novelty* (uses as fitness score).

10.4 Quality-Diversity Optimisation

1. Take **stochasitic selection** of solutions. *Can be biased towards quality/novelty.*
2. Offspring generated via mutation/crossover.
3. Each offspring is evaluated for **quality** (*fitness*) & **behavioral descriptor**.
4. Offspring is added to collection if *more novel* or *higher quality*.

Multidimensional Archive of Phenotypic Elites (MAP-Elites) is a QD algorithm to discretise the behavioral desc space in a grid, then try to fill it with the best solutions. Each new solution fills a cell corresponding to the behavioral desc. If cell already occupied, replace existing solution if we have higher fitness. *Grid size is hyperparam.* Easy to impl, but density not uniform.

10.5 Quantifying QD Performance

Can use **diversity** (*archive size*), **performance** (*max/mean fitness val*) & **convergence speed** of both metrics. Summarized as **QD-Score**: *sum of fitness of all solutions in archive*.