Overview

ML:
Given unknown target function f, hypothesis function h & data D $\rightarrow$ try to approximate the unknown target function using the data (want hypothesis function to give same output as target for given input data)
Assumes data D generated from distribution of unknown function
Supervised Learning:
Input with Input Features + Correct Output Labels $\rightarrow$ Model Generates Estimated Output Label
Unsupervised Learning:
Input with Input Features (No Output Label) $\rightarrow$ Model tries to find Patterns in Data & Estimate Output Label (i.e. lossy data compression)
Reinforcement Learning:
Input with Input Features (No Output Label) $\rightarrow$ Reward Signal Fed Back to Learning Algorithm
Classification: (Supervised) Input $\rightarrow$ Output (Discrete/Categorical Label)
Binary Classification: each instance assigned one of 2 classes
Given a Dataset - Transform into a Feature Space
Linear Classifier: learns a linear function between two spaces (efficient & simple to implement - good when classes linearly separable - may not be capable of modelling complex relationships)
Multi-Class Classification: each instance assigned one of 2+ classes
Multi-Label Classification: each instance assigned a set of target labels
Regression: (Supervised) Input $\rightarrow$ Output (Real, Continuous Variable)
Simple Regression: single-variable input $\rightarrow$ single-variable output
Multiple Regression: multi-variable input $\rightarrow$ single-variable output
Multivariate Regression: multi-variable input $\rightarrow$ multi-variable output
Supervised Learning Pipeline:
Training Dataset D generated from unknown true function $f(X)$ & gives instance $\{X^{(i)}, Y^{(i)}\}$ & split into $X^{train} \& Y^{train}$
X^{train} data passes through a feature encoder & transformed to feature space
Feature encoded inputs & labels as input to algorithm produces model $h(X)$
Model takes X^{test} & produces output which can be evaluated against Y^{test}
Feature Encoding:
Feature Encoded Instances $X^{(i)}$ given by K-dimensional vector of features
Each feature ($x_k^{(i)}$) represents a dimension in the K-dimensional Feature Space
Features can be categorical/integers/real-valued numbers (& may normalise)
Feature Scaling: Standardisation: $x_k^i = x_k^{(i)} - \mu_k / \sigma_k$
Normalisation scales features to a similar range - no feature dominates & each feature contributes approximately proportionately to the final prediction
Curse of Dimensionality:
More Features $\rightarrow$ Higher Dimensional Feature Space
Increased Computational Complexity: Computation (for algorithm) increases exponentially with increased dimensions & simpler algorithms may become impractical
Data Sparsity: as increase no. dimensions - volume of space increases exponentially & need more data points to fill space (difficult for algorithm to learn on sparse data)
Overfitting: with many features & not enough instances - model likely to fit the training data too closely & capture noise as if it were a real pattern - giving poor generalisation to unseen data
Mitigations:
Feature Selection: only select a subset of important features
Dimensionality Reduction: reduce dimensionality of data while preserving essential characteristics
Feature Engineering (Manual Feature Extraction): combine/transform existing features to create new ones that capture relevant information in fewer dimensions
Automatic Feature Extraction: use statistical methods to analyse what features are more important to represent the data
Feature Learning (without Feature Encoding):
Advantages:
Reduced need for manual feature engineering (discover non-obvious patterns)
Reduced Bias & Human Error (prevents manual biases introduced)
Saves Time & Effort
Adaptability (models can adapt to complexities without requiring extensive domain knowledge for feature engineering)
Generalisation (learned features may be able to better generalise to unseen data as they capture intrinsic patterns more effectively)
Limitations:
Requires Large Datasets (to learn features effectively & not overfit)
Increase Computational Cost (algorithm takes more of the work)
Risk of Overfitting (models may overfit the training data - if they are learning features in a high-dimensional space)
Loss of Interpretability (more complex & less interpretable model)

ML Algorithms

Lazy Learner: does not build a generalisable model during training - waits until a new data point is presented at test time & uses the entire dataset to generate predictions (delays computation until test time)
Eager Learner: builds a generealisable model during training - learning the structure that best fits the training data - once trained does not need training data anymore (most computation done during training)
Lazy Learner $\rightarrow$ Longer Test Time & Eager Learner $\rightarrow$ Longer Train Time
Non-Parametric Model: does not make strong assumptions about the structure of the function that relates input features to output predictions (no fixed number of parameters - complexity can grow with amount of training data)
Nearest Neighbour: (Lazy) at test time - looks for the closest data to test data & classifies according to that data (that data could be noise)
Linear Model: (Eager) assumes data linearly separable (separable by N-1 dimensional hyperplane) & finds the best linear function to separate data
Nonlinear Model: (Eager) data linearly non-separable & find best non-linear function to separate data (transform feature space & make data linearly separable - classifying with linear model) (classify using multiple linear models)

Overfitting & Underfitting

Overfitting: learns training data too well (including noise & outliers) rather than capturing underlying pattern (poor generalisation)
Overfitting Causes:
Model too Complex (too much capacity to learn fine details of training data)
Training Set not representative of True Distribution (i.e. small dataset)
Too Long Training Process / Noisy Training Data / Lack of Regularisation
Overfitting Mitigations:
Use Correct Level of Model Complexity (tune on validation set)
Regularisation / Cross-Validation / Early Stopping
Underfitting: performs poor on training data & poor generalisation
Underfitting Causes: Overly Simple Model / Insufficient Training / Poor Feature Selection
Underfitting Mitigations: Increase Size of Dataset to get True Distribution
Increase Complexity / Perform Training to Longer (more Epochs)
Feature Engineering (transform/add features to better represent distribution)
Overfitting in NNs: NNs with enough capacity (no. of parameters) can easily overfit
Higher Capacity $\rightarrow$ More Trainable Parameters $\rightarrow$ Model can Memorise Data rather than Learning Patterns in Underlying Distribution
Overfitting Mitigations in NNs: (Reduce Capacity or Acquire More Data)
Early Stopping $\rightarrow$ use Validation Data to choose optimal point to stop training (point at which accuracy begins to drop) (in practice - evaluate on validation data every epoch & always store best model so far - stopping when performance not improved over a no. of epochs - revert to best model so far) (avoids overfitting to training datapoints)
Regularisation $\rightarrow$ adding information/constraints to prevent overfitting (e.g. penalising how large weights can be $\rightarrow$ reducing capacity)
L2 Regularisation $\rightarrow$ + squared weights to loss (penalising larger weights - prefers many small weights & encourages feature sharing - less sensetive to outliers & provides stable solutions)

New Loss: $J(\theta) = \text{Loss}(y, \hat{y}) + \lambda \sum_w w^2$ and $w \leftarrow w - \alpha \left(\frac{\partial \text{Loss}}{\partial w} + 2\lambda w \right)$

L1 Regularisation: $\rightarrow$ + absolute weights to loss (produces sparse models as encourages weights to be 0 - can be seen as automatic feature selection - keeps only most important features)

$$J(\theta) = \text{Loss}(y, \hat{y}) + \lambda \sum_w |w| \quad w \leftarrow w - \alpha \left(\frac{\partial \text{Loss}}{\partial w} + \lambda \text{sign}(w) \right)$$

Dropout: in training - randomly set some neural activations to 0 (typically 50% of activations in layer) - encourages learning more features
Form of Regularisation: prevents NN from relying on any particular feature - as there is a 50% probability that the feature will not in NN
$\rightarrow$ becomes harder to overfit to edge cases in training data & learns better generalisable model of feature space
During Testing: all of the neurons used (but activation must be scaled) $\rightarrow$ when neurons deactivated with dropout - expected sum of the activations changes & scaling the activations becomes necessary to compensate for this reduction (if do not scale - during testing the NN can become overly activated & distort behaviour - scaling factor inversely proportional to dropout rate)
Form of Ensemble Learning: each training iterations effectively trains a different subnetwork & during testing the full network is active - averaging the predictions made by subnetwork (makes NN more robust & generalisable)

Bias-Variance Tradeoff

Bias:
Systematic Error of model due to some unfounded assumptions made about relationship between input features & outputs
Difference between average prediction & ground truth
High Bias: model is too simple too approximate complex relationship between input features & outputs
(model is biased towards preferentially detecting certain types of relationships between input features & outputs) (training error)
Variance:
Quantifies the sensitivity of the model to the training set used
Quantifies the variability of the models predictions for the same input across different training sets
High Variance: model does everything it can to perfectly map the input features to outputs in the training set
(giving all attention to one particular training set for deriving relationship between input features & outputs) (difference between train & testing error)
Bias-Variance Tradeoff:
High Bias: prone to underfitting & model to simple to accurately captures underlying patterns in data
High Variance: prone to overfitting & model performs well on specific training set - failing to generalise to new unseen data
(tied too closely to training set & captures noise in that set as if it were a real pattern to approximate outputs from input features)
Optimal: low bias & low variance (in practice - middle ground)
Low Variance & High Bias: model overly simplistic
High Variance & Low Bias: model overly complex
High Variance & High Bias: model fails to fit training data & fails to generalise to unseen data

kNNs

Instance-Based Learning: makes predictions or decisions (inference) based on individual examples/instances of the training data (rather than constructing a generalisable model - relies on memorising the training instances & using them directly for making predictions - lazy learner)
Advantages (Instance-Based):
Adaptable (can adapt quickly to changes in data as does not rely on fixed model)
No Training Phase (no training as solely makes inference at test time)
Simple Implementation (simply reasoning on training data at test time)
Suitable for Non-Linear Data & Captures Complex Relationships
Disadvantages (Instance-Based):
Memory-Intensive (storing all training instances)
Slow Inference (calculating similarities for every new instance - time-consuming & high computational cost)
Sensitive to Noise (relies on individual instances)
Susceptible to Curse of Dimensionality
Sensitive to Irrelevant Features
Difficulty in Capturing Global Trends (relies on nearby instances)
k-Nearest Neighbours Classifier:
Single Nearest Neighbour: sensitive to noise & overfits training data
kNN (solution): infers from k-nearest samples & classifies according to largest number of samples (k typically odd)
kNN Advantages (+above):
Flexible in Distance Choice & used for Classification & Regression
Robust to Noise (for » k)
kNN Disadvantages (+above):
Curse of Dimensionality (as number of features increases - feature space becomes sparser - making it difficult for kNN to find nearest neighbours effectively) - Solution: Weight Features / Perform Feature Extraction
Parameter (k) selection significantly impacts bias/variance & results
Requires Feature Scaling (Normalisation/Standardisation)
Selecting k (hyperparameter):
Increasing k $\rightarrow$ Increases Bias (smoother decision boundary) & Decreases Variance (less sensitive to training data)
Appropriate Distance Metric must be selected (L1/L2(Euclidean)/L∞)
Distance-Weighted kNN: assigns weights to each neighbour (higher for closer samples) & sums weights per class in neighbourhood
Weights $w^{(i)}$ could be Inverse of Distance/Gaussian Distribution
Less sensitive to hyperparamter k (distance examples $\rightarrow$ less weight)
Robust to Noisy Training Data - weighted combination of neighbours smooths out impact of isolated noise (improved performance)
kNN for Regression: ompute the mean value across k-nearest neighbours
Locally Weighted Regression: distance-weighted kNN for Regression

Decision Trees

Decision Tree: (Eager) model in which each internal node represents a decision on an input feature - each branch representing the outcome of the test - & each leaf node representing a class label (in classification) or numeric value (in regression)
Algorithm (ID3):
Search for the Optimal Splitting Rule on Training Data (maximising IG) & Split Dataset on Optimal Splitting Rule
Repeat 1 & 2 until a Stopping Criterion is met (i.e. max tree depth, min no. samples in node, further splitting does not significantly improve performance, or subset pure)
Advantages:
Interpretability (decision trees easy to understand & visualise) & Versatility (can be applied to both Classification & Regression)
Non-Linearity (captures non-linear relationships) & Robustness (can handle numerical/categorical data without extensive prepossessing)
Limitations:
Overfitting (prone to overfitting - especially if they are allowed to grow/dataset is noisy)
Instability (small changes to data can lead to different tree structures) & Bias (biased if certain classes more prevalent in training data)
$\rightarrow$ **succession of linear decision boundaries that classify the dataset**
Selecting the Optimal Splitting Rule:
Select the subsets that are as **pure** as possible (pure dataset: only has samples of a single class) $\rightarrow$ using Information Gain
Information Entropy:
Measures Uncertainty of a Random Variable (average amount of information required) (high entropy $\rightarrow$ more information)
Information Required to Fully Determine State: $I(x) = -log_2(P(x))$

Entropy: $H(X) = -\sum_{k=1}^{K} P(x_k) \log_2 P(x_k)$ / $H(X) = -\int f(x) \log_2(f(x))\, dx$

Information Gain: Entropy of Entire Dataset - Weighted Average Entropy of Subsets
Weighted Average Entropy of Subsets: sum of the entropy of each subset weighted by the cardinality (size) of each subset
Categorical/Symbolic Values: (classification: no. categories = no. elements in Entropy Sum)
Calculate Entropy for Entire Dataset H(D)
Calculate IG for Each Feature as if Splitting on that Feature, by Finding Entropy of each Subset (subset: set with a particular value of that feature) & subtracting Weighted Sum of Subset Entropys from Total Entropy of that Set
Select Feature with Highest IG & Split on all Possible Values of that Feature - Repeat for Split Subsets & Remaining Features until Criterion
Sanity Check: When Calculating Entropy - Probabilities Sum to 1
Continuous Values:
Sort all samples for a given feature - select the midpoint between each pair of samples & calculate the IG if dataset split on this midpoint - repeat for each feature & selecting splitting rule that gives maximum IG

Evolutionary Algorithms (1)

Genetic Algorithms:
Initialisation $\rightarrow$ Population $\rightarrow$ Selection $\rightarrow$ Parents $\rightarrow$ Offspring $\rightarrow$ New Population $\rightarrow$ Evaluation $\rightarrow$ Converged?
Initialisation: initial population of potential solutions is created (consists of individuals each with a solution represented by a genotype)
Selection: individuals are chosen based on their fitness (the selection operator is used to select the fittest individuals to create offspring)
Parents: selected individuals that will undergo crossover & mutation to produce offspring
Offsprings: new individuals created from parents, possessing a mixture of their genotypes + possibly some mutations
Evaluation: genotypes (encoded solutions) are developed in phenotypes (actual solutions) that can be evaluated by a fitness function

Genotypes: potential solutions encoded as a binary string
Phenotypes: expressed solution of the genotype - actual solution to problem
Fitness Function: used in evaluation to evaluate fitness of each individual (represents the problem to optimise)
Selection Operators: based on their fitness - individuals are selected to contribute their genes to next generation
Cross-Over Operators: pairs of individuals chosen to undergo crossover - parts of genotypes are exchanged & producing new individuals (single-point crossover: split point randomly picked - genotype of offspring formed by exchanging portions of parents genotype)
Mutation Operators: with small probability - random changes introduced in offspring's genotype (to avoid premature convergence to suboptimal solutions) (standard mutation: for each bit in the genotype - a number between 0 & 1 is randomly generated (uniform distribution) - if this number lower than 1/size of genotype - genotype bit flipped)
Iteration: Selection $\rightarrow$ Mutation repeat over many generations - evolving population & increasing fitness
Termination: terminates when condition is met (i.e. max. number of generations/fitness level/plateau in fitness)

Evaluation

Setup: (for sufficiently large datasets)
Shuffle Dataset: avoids splitting similar data into the test dataset
Split: into training set, (validation set) & held-out test set
Evaluation: held-out test set provides unseen data that model should generalise to - used to evaluate model & give accuracy value
Hyperparameter Tuning:
Hyperparameters are chosen before training & fixed during training process (vs Model Parameters that are optimised during training)
To find optimum hyperparameter values that generalise to unseen data - split dataset into training, validation, & testing sets (e.g. 80/10/10)
Try different hyperparameter values on the training dataset (multiple models) & select best model according to accuracy on validation dataset (evaluating how model generalises) - performing final evaluation on test dataset (unseen)
(if hyperparameter values selected based on training set accuracy - would overfit to training data & not generalise) (if hyperparameter values selected on test set - test set no longer held out & becomes part of training process)
Final Training:
After best hyperparameters selected with validation set - can train best model again using training + validation sets to extract further performance
Initial Training $\rightarrow$ Hyperparameter Tuning $\rightarrow$ Final Train $\rightarrow$ Testing
Cross-Validation: (for small datasets)
For small datasets - splitting into training/validation/test not possible (test will not have enough samples to evaluate performance) $\rightarrow$ use cross-validation
Divide dataset into k (e.g. 10) equal folds & use k-1 for training + validation & 1 for testing
Iterate k times - each time testing on a different portion of the data
Performance on all k held-out test sets can be averaged
Evaluates algorithm performance rather than a particular model - as training datasets change on each fold (new model)
Entire dataset used for testing (without using the same data for training & testing ant any one time) - giving enough examples to evaluate performance

Global Error Estimate: $\frac{1}{N} \sum_{i=1}^{N} e_i$

Cross-Validation Hyperparameter Tuning:
(1) Extend Splitting: Add 1 Rotating Fold for Validation & use to Tune Hyperparameters (finds different optimal parameters in each fold - so only using a small dataset for tuning of each model - may not be representative)
(2) Cross-Validation in Cross-Validation: Performs cross-validation with separate testing folds & within each cross-validation - performs an internal cross-validation over the remaining folds with the validation dataset
Allows tuning of the best hyperparameters for each testing fold (using entire dataset rather than single fold)
Each testing fold gives different hyperparameters & so each model has different hyperparameters & testing data
Take the model for each testing fold with the optimal hyperparameters & perform final evaluation of each model - take average to evaluate algoirthm performance (high computation but yields best results)
Production: once hyperparameters tuned - can train model again using entire dataset (trade-off: extract extra performance vs having held-out test set to evaluate final performance)

Evaluation Metrics

Confusion Matrix: visualisation to analyse model performance
Binary Classification Confusion Matrix: each row represents an actual label/class & each column represents a predicted label/class (providing insight to improve model performance) [TP, FN] [FP, TN]
Accuracy: Number of Correctly Classified Examples / Total Number of Examples in Dataset (TP + TN / TP + TN + FP + FN)
Classification Error = 1 - Accuracy
Precision: Number of Correctly Classified Positive Examples / Total Number of Predicted Positive Examples (TP / TP + FP)
High Precision Model: if it predicts a label positive $\rightarrow$ confident label is actually positive (may not be finding all positive examples)
Recall: number of correctly classified positive examples / total number of positive examples (TP / TP + FN)
High Recall Model: good at retrieving positive examples from dataset (may be labelling many negative examples as positive)
Precision-Recall Tradeoff: optimising for recall often lowers precision & optimising for precision often lowers recall
F-Measure: Precision & Recall into a single score
F_1 - equal importance to both & $F_{0.5}$ precision 2x important as recall
$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$ $F_\beta = \frac{(1+\beta^2) \cdot \text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$
Macro-Averaging: Recall, Precision, & F-Measure computed for individual classes - MA finds average across all classes (average on class level)
Micro-Averaging: average on item level (micro-averaged P,R,F1 = accuracy for binary & multi-class classification)
Multi-Class Classification: Precision, Recall, & F1 computed for each class separately (setting one class as positive) & then finding macro-average
All Evaluation Metrics + Confusion should be considered together
Regression - MSE:

MSE: $\frac{1}{N} \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2$ (average how far predictions are from actual values)

RMSE: $\sqrt{MSE}$ (transforms units of evaluation metric to same as data values)
MSE vs RMSE: MSE more sensitive to outliers - MSE differentiable & RMSE not differentiable at 0 - MSE does not have same units as data & RMSE does (more interpretable)
Other Metrics: Accurate, Fast, Scalable, Simple, Interpretable
Imbalanced Data Distributions:
Balanced Dataset: number of examples in each class similar - all measures result in similar performance for each class
Imbalanced Dataset: classes not equally represented - accuracy reduced & P/F1 are significantly less for the underrepresented class
Accuracy can be misleading (follows performance of majority class) - Macro-Averaged Recall helps detect if one class completely misclassified (but no information about FPs) - F1 useful (but can also be affected by imbalanced dataset) - **Consider all Evaluation Metrics & Confusion Together**
Solutions to Imbalanced Datasets:
Normalise Counts: divide TP/TN/FP/FN by total number of examples per class such that confusion matrix rows sum to 1
Downsample Majority Class: select randomly same number of examples as minority class
Upsample Majority Class: create duplicates until both classes same size
(helps balance datasets but results do not reflect how well model generalised - as real data imbalanced)
Confidence Intervals:
Quantifies confidence in evaluation result (confidence that result from model matches underlying data distribution) (larger test set $\rightarrow$ higher confidence)
True Error: probability that model (h) will misclassify a randomly drawn example x from distribution D (difference between the expected prediction of the model and the true value - cannot calculate)
Sample Error: based on model (h) & sample (S) (difference between prediction of model based on sample & true value - can calculate)
(= classification error on test dataset = 1 - accuracy)
Confidence Interval: an $N\%$ Confidence Interval for some parameter q is an interval that is expected with probability $N\%$ to contain q

$\text{error}(h) \pm Z_N \sqrt{\frac{\text{error}_s(h) \cdot (1 - \text{error}_s(h))}{n}}$ (Z_N: Scaling Factor for Confidence)

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

Testing for Statistical Significance:
Statistical Tests: if the means of two sets a significantly different
Randomisation Test: randomly switch some predictions from both models & measure how often the new performance difference is greater than or equal to 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 matches results (e.g. classification error over same folds in cross-validation)
Tests tell us if the performance difference is due to random sampling or is more statistically significant
p-value: probability that given a performance difference - the models actually perform the same & the performance difference is not true (due to sampling error)
For a small p-value (¡0.05) can be confident that performance difference is actually true (performance difference is statistically significant)
p ¿ 0.05 $\rightarrow$ cannot observe a statistical difference (does not mean algorithms are similar)
P-Hacking: misuse of data analysis to find patterns in data that can be presented as statistically significant when in fact there is no underlying effect (when running large number of experiments - $p < 0.05$ gives some significant results that will be false positives)
P-Hacking Mitigation: Adaptive Threshold for p-Value
Rank p-values from M Experiments ($p_! \leq p_2 \leq \ldots \leq p_m$)
Calculate Critical Value for Each Experiment ($z_i = 0.05 \frac{i}{M}$) i - index
Significant Results are the ones with p-values smaller than critical value
Much Stricter than Original Threshold (0.05) - allows much larger number of experiments (with confidence that findings are actually statistically significant & not False Positives)

Linear Regression

Simple Linear Regression: single input variable (x) - $y = ax + b$
Sum-of-Square Loss Function: (quadratic - large E = exponential penalty)

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

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

Repeat & optimise each parameter in model (during model training)
Gradient Descent: (Optimising Linear Regression)
Repeatedly update parameters a & b by taking small steps in negative direction of partial derivative of loss function w.r.t to parameter

$$a := a - \alpha\frac{\partial E}{\partial a}, \quad b := b - \alpha\frac{\partial E}{\partial b}$$

$$a := a - \alpha\sum_{i=1}^{N}(ax^{(i)} + b - y^{(i)})x^{(i)}, \quad b := b - \alpha\sum_{i=1}^{N}(ax^{(i)} + b - y^{(i)})$$

α : Learning Rate (hyperparmeter to be tuned)
Iterate through dataset & calculate new parameters a & b $\rightarrow$ taking small steps in negative direction of partial derivative
Epoch: each iteration/pass through the dataset (multiple epochs)
$\rightarrow$ can keep features & outputs in vectors & use vector operations

Gradient: $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \left(\frac{\partial f(\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_2}, \ldots, \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_K}\right)^T$

Analytical Solution:
$\mathbf{X} = [x^{(i)}, 1]_{i=1}^{N}, \mathbf{y} = [y^{(i)}]_{i=1}^{N}, \boldsymbol{\theta} = [a, b]^T, \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) = \mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = 0$
$\boldsymbol{\theta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$
Not Practical for Large Datasets (Significant Computation & Inefficient)
Multiple Linear Regression: (multiple input features)
$y^{(i)} = \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \theta_3 x_3^{(i)} + \ldots + \theta_K x_K^{(i)} + \theta_{K+1}$

Neural Networks

Neuron:
Features x: $\begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}^{\mathsf{T}}$ Weights W: $\begin{bmatrix} \theta_1 & \ldots & \theta_n \end{bmatrix}^{\mathsf{T}}$ Activation: g
Output: $\hat{y} = g(\theta_1 x_1 + \ldots + \theta_n x_n + b) \rightarrow \hat{y} = g(W^T x)$
Activation Function applied to Linear Regression Equation
To make b part of Features/Weights $\rightarrow$ add Feature x_{n+1} & Weight θ_{n+1}
Logistic Activation Function (Sigmoid): $g(z) = 1/(1 + e^{-z}), \ \hat{y} \in [0, 1]$
Logistic Regression: passes linear regression output through sigmoid
$\rightarrow$ used for Binary Classification (fits points effectively to 0 or 1)
Perceptron: (supervised binary classification algorithm)

$h(x) = f(\mathbf{W}^{\mathsf{T}}x) = \begin{cases} 1 & \text{if } \mathbf{W}^{\mathsf{T}}x > 0 \\ 0 & \text{otherwise} \end{cases}$ 1 if Product of Weights & Features ¿ 0

Perceptron Learning Rule: $\theta_i \leftarrow \theta_i + \alpha(y - h(x))x_i$
(α : Learning Rate, y: Desired Output, $h(x)$: Predicted Output, x_i: Input)
If Prediction Correct $(y = h(x)) \rightarrow$ Weight Stays the Same
If $y = 1$ & $h(x) = 0 \rightarrow$ magnitude of weight increased & $W^T x$ increases
If $y = 0$ & $h(x) = 1 \rightarrow$ magnitude of weight decreased & $W^T x$ decreases
$\alpha \rightarrow$ hyperparameter to be tuned during validation (learning rate)
Perceptron used on only linearly separable functions & not differentiable
Multi-Layer NNs: model more complex relationships (each neuron connected in parallel becomes a feature detector & detects a different feature)
Layers: (input - hidden - output) each layer learns higher order features & output of each layer passed to the next (can use different activations)
Traditional Patter Recognition (before NNs): Input Data $\rightarrow$ Manually Crafted Feature Extractor $\rightarrow$ Trainable Classifier $\rightarrow$ Output
End-to-End Training (NNs): Input Data $\rightarrow$ Trainable Feature Extractor $\rightarrow$ Trainable Classifier $\rightarrow$ Output (allows NN to learn useful features itself with lower levels acting as trainable feature extractors & higher levels acting as trainable classifier - which are trained jointly & updated simultaneously)
Activation Functions: introduce nonlinearity into multilayer NNs
Activation $\rightarrow$ transforms output of Neuron before passing to next layer
Linear Activation: directly passes output of linear layer (i.e. no activation)
$f(x) = x, \ \hat{y} = W^T x$ - Multi-Layer NN collapses to Single-Layer
Sigmoid Activation: compresses output smoothly into the range between 0 & 1 $\rightarrow f(x) = \sigma(x) = 1/(1 + e^{-x})$
Tanh Activation: similar to $\sigma(x)$ but ranges between -1 & 1
$\rightarrow f(x) = tanh(x) = (e^x - e^{-x}/e^x + e^{-x})$
ReLU Activation: linear in the positive part & non-linear overall

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

Softmax Activation: scales the inputs into a probability distribution (largest input will be large & rest small) - all output values sum to 1
$\rightarrow \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$
$\rightarrow$ used for multiclass classification (assigning most probability to class most confident on)
Most Activation Functions applied element-wise (apart from Softmax)
ReLU commonly used for hidden layers & tanh/sigmoid can be more robust
Choice of Activation Function $\rightarrow$ Hyperparameter (can be set or tuned)
Loss Functions: function to minimise (better performance $\rightarrow$ lower loss)

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

Binary Cross-Entropy Loss:
$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right]$$

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

Typical Combinations:
Regression: Linear Output Activation - MSE Loss Function
Binary Classification: Sigmoid Output Activation - Binary Cross-Entropy Loss (e.g. stock prices up or down)
Multi-Class Classification: Softmax Output Activation - Categorical Cross-Entropy Loss (e.g. Image Recognition)
Multi-Label Classification: Sigmoid Output Acitvation (0 or 1 for each label $\rightarrow$ each label has own sigmoid activation) - Binary Cross-Entropy Loss (each output neuron acts as own binary classifier for each label)

Forward Pass:
Forward Pass: calculating the output of the network for a given input
Feeds Data into Network - Passes Data Through Hidden Layers performing Linear Transformation & Non-Linear Activation - Processed Data Reaches Output & Gives Prediction - Calculates Loss to Evaluate Performance
Forwards: Input $\rightarrow$ Linear Layer $\rightarrow$ Activation Function $\rightarrow$ Next Layer

Evolutionary Algorithms (3)

Novelty Search: optimises novelty rather than quality (prioritising diversity over direct performance) to explore solution space
Novelty$(x) = \frac{1}{N}\sum_{k=0}^{N} d_i(x)$ (average distance between individual x & N nearest neighbours)
Behavioural Descriptor: characterises certain aspects of the solutions (defining types of solutions/combinations of features)
Replaces Fitness Function with **Novelty Score** & adds most Novel Individuals to a Novelty Archive (used in evaluation)
Quality Diversity Algorithms: learn in a single optimisation process a large collection of diverse & high-performing solutions
Stochastic Selection $\rightarrow$ Random Mutation $\rightarrow$ Evaluation $\rightarrow$ Tentative Addition to Collection
Measures Performance with Fitness Function & Diversity with Behavioural Descriptor
Map-Elites: QD-Algorithm that discretises the behavioural descriptor space in a grid & tries to fill with best solutions
Grid: sets of cells representing behavioural descriptor space - 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 solution with best fitness is kept
Size of Cells $\rightarrow$ Hyperparameter - Advantage $\rightarrow$ Easy to Implement - Limitation $\rightarrow$ Density may not be Uniform
Grid + Uniform Random Selection (Grid Initialised with Randomly Generate Solutions & Mutations Randomly Selected added)
Metrics: Diversity of Solutions in Container: Archive Size - Performance of Solution : Max/Mean Fitness Value - Convergence
QD-Score: sum of fitness of all solutions **QD Algorithm Selector:** selects individual to mutate & join next generation (uniform)

Backpropagation

Mini-Batching: combining vectors of several datapoints into one
During Training - model parameters are updated after processing each batch
Advantages: improves speed (GPUs can parallelise matrix multiplication operations) - reduces noise (calculating gradient for multiple datapoints reduces influence of noise in a single point & improves generalisation of model/reduces overfitting) - reduces memory utilisation (memory for weights shared across multiple instances - more efficient memory use)
Batch Size: hyperparameter that can be tuned
Small Batches: offer regularising effect & more noise in gradient estimation
Larger Batches: more accurate estimation of gradient & more memory/computational power
Backwards Pass: adjusting parameters of the network (backpropagation)
Calculates Gradient of Loss Function w.r.t Output at Output Layer - Propagates Gradients back through the Network (layer-by-layer) - uses Gradients to Perform Gradient Descent & Update Parameters
$\frac{\partial Loss}{\partial \hat{Y}} \rightarrow \frac{\partial Loss}{\partial A^{[2]}} \rightarrow \frac{\partial Loss}{\partial A^{[1]}} \rightarrow \frac{\partial Loss}{\partial X}$
Backpropagation: Neuron $Z = XW + B$
$\partial Loss/\partial Z$ from Layer Above $\rightarrow \partial Loss/\partial W$(Update) & $\partial Loss/\partial X$(Propagate)
$Z : N \times M$ (number of datapoints $\times$ number of neurons in layer)
$X : N \times D$ (number of datapoints $\times$ number of features)
$W : D \times M$ (number of features $\times$ number of neurons in layer)
$B : N \times M$ (number of datapoints $\times$ number of neurons in layer)
Consider: $X \rightarrow Z^{[1]} = XW^{[1]} + B^{[1]} \rightarrow A^{[1]} = g_h(Z^{[1]}) \rightarrow Z^{[2]} = A^{[1]}W^{[2]} + B^{[2]} \rightarrow \hat{Y} = g_o(Z^{[2]})$
Gradient at Output: $\frac{\partial L}{\partial \hat{Y}}$
Applying Activation Function: $\frac{\partial Loss}{\partial Z_2} = \frac{\partial Loss}{\partial \hat{Y}}\frac{\partial \hat{Y}}{\partial Z_2} = \frac{\partial Loss}{\partial \hat{Y}} \circ g'(Z_2)$
Gradient to Update Weights: $\frac{\partial Loss}{\partial W_2} = \frac{\partial Loss}{\partial Z_2}\frac{\partial Z_2}{\partial W_2} = A_1^T \frac{\partial Loss}{\partial Z_2}$
Gradient to Update Biases: $\frac{\partial Loss}{\partial B_2} = \frac{\partial Loss}{\partial Z_2}\frac{\partial Z_2}{\partial B_2} = 1^T \frac{\partial Loss}{\partial Z_2}$
Gradient to Propagate: $\frac{\partial Loss}{\partial A_1} = \frac{\partial Loss}{\partial Z_2} W_2^T$
Applying Activation Function: $\frac{\partial Loss}{\partial Z_1} = \frac{\partial Loss}{\partial A_1}\frac{\partial A_1}{\partial Z_1} = \frac{\partial Loss}{\partial A_1} \circ g'(Z_1)$
...
(Element-Wise Multiplication - Applied to Activation apart from Softmax)
Derivative of Activation Functions:
Linear: $g(z) = z \quad g'(z) = 1$
Sigmoid: $g(z) = 1/(1 + e^- z) \quad g'(z) = g(z)(1 - g(z))$
Tanh: $g(x) = (e^z - e^{-z})/(e^z + e^{-z})$
ReLU: $g(z) = \begin{cases} z & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}, \quad g'(z) = \begin{cases} 1 & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases}$
Softmax: (Joint Derivative of Softmax & Cross-Entropy Loss) $\frac{\partial L}{\partial z} = (y - \hat{y})$
Derivatives of Loss Functions:
MSE: $\frac{\partial \text{MSE}}{\partial \hat{y}_i} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$
BCE: $\frac{\partial \text{BCE}}{\partial \hat{y}_i} = -\frac{1}{n}\sum_{i=1}^{n}\left[\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}\right]$ **CCE:** $\frac{\partial \text{CCE}}{\partial \hat{y}_{ij}} = -\frac{y_{ij}}{\hat{y}_{ij}}$

Gradient Descent

Gradient Descent: repeatedly updating model parameters by taking small steps in the negative direction of the partial derivative of the loss function $\rightarrow$ model better at predicting data at each iteration (minimising loss function)
$W = W - \alpha\frac{\partial L}{\partial W}$ (α: learning rate/step size - hyperparamter to be tuned)
Loss Landscape: (spans parameter space) - gradient descent aims to find global minimum in loss landscape (may converge at local optimum depending on starting point) (all activation & loss functions must be differentiable)
Algorithm: Initialise Weights Randomly - Loop until Convergence (minimised loss function) - Compute Gradient - Update Weights - Finish
Batch Gradient Descent: uses the entire dataset to compute the gradient o the loss function for a single update of model parameters (very accurate gradient, stable & convergent - suitable for small datasets) (extremely computationally intensive for larger datasets - memory demanding - more likely to become stuck in local minima)
Stochastic Gradient Descent: uses a single datapoint to compute the gradient of the loss function for a single update of model parameters (faster updates & convergence - computationally efficient - suitable for large datasets as memory efficient - can escape local minima more easily due to noise) (very noisy gradient estimations leads to less stable convergence - more likely to not reach minima/fluctuate)
Mini-Batch Gradient Descent: uses small batches of datapoints to compute the gradient of the loss function for a single update of model parameters (compromise - moderate level of noise/relatively stable & escapes local minima - balances stability of full-batch & stochastic nature - more computationally efficient than stochastic & memory-efficient than full-batch)
$\rightarrow$ Limitation: Mini-Batch Size is a Hyperparameter that requires tuning
Learning Rate Importance: if learning rate too low $\rightarrow$ model will take too long to converge & if learning rate too high $\rightarrow$ will keep stepping over the optimal values
Sufficient Learning Rate: finds optimum in reasonable number of steps
Adaptive Learning Rate: different learning rate for each parameter (takes bigger steps if a parameter has not bee updated much recently & take smaller steps if a parameter has been getting many big updates)
Learning Rate Decay: Scaling Learning Rate by a Value [0,1] such that Rate becoming Smaller (simple Adaptive Learning Rate)
Update Rule: $\alpha \leftarrow \alpha d \quad d \in [0, 1]$
Take Smaller Steps as we get closer to the minimum (such that we do not overshoot & still converges in reasonable no. steps)
Strategies for performing the Update (Decay):
Every Epoch / After a Certain Number of Epoch / When Performance on Validation Set has not Improved for Several Epochs
Weight Initialisation: (Initialising W before Training)
Zeros: common to set the biases to 0 (do not want Neurons to start biases) - if all weights are set to 0/the $\rightarrow$ many of the neurons will learn the same pattern as they receive the same information during backpropagation & will be optimised to learn the same values
Randomly: initialise weights randomly (neurons incentivised to learn different information)
Normal: draw randomly from a normal distribution (i.e. mean 0 & variance 1 - giving small values around 0)
Xavier Glorot: draws values from a uniform distribution (many neurons on either side of layer $\rightarrow$ initialised with smaller values close to 0)
Randomness in Network: different random initialisation of weights leads to different results (different minimum of loss function)
Solution: allow randomness $\rightarrow$ run with different random seeds & report average (could set explicitly the same seed $\rightarrow$ but GPU Threads finish in a random order & introduce randomness anyway)
Data Normalisation/Feature Scaling: optimise NNs by normalising input data (crucial for effective training - typically performed by a preprocessor before Training Process)
Min-Max Normalisation: rescale range between an arbitrary set of values [a,b] (scaling the smallest value to a & largest to b - often 0 & 1)
$X' = a + \frac{(X - X_{min})(b-a)}{X_{max} - X_{min}}$
Advantages: all features brought to same scale - linear transformation maintains relationship between points - easy to implement
Disadvantages: sensitive to outliers - data not centered on 0 - distorts original representation of data - dependent on choice of min-max
Standardisation (z-Normalisation): scales the data to have mean 0 & standard deviation 1
Advantages: less sensitive to outliers - centered around 0 - preserves original data distribution (with a scaling factor) - consistent scale
Disadvantages: if data has non-linear features may not preserve relationships between variables as effectively as min-max normalisation
Without Feature Scaling: if features have different scales $\rightarrow$ gradients may be disproportionately influenced by certain features (as different sizes steps taken for different features) - may be difficult to find a suitable learning rate for all scales of input features - loss function can become elongated & narrow (making path of gradient descent more complex & potentially leading to longer training times/stuck in local minima)
With Feature Scaling: if all features in a similar range & proportional to weights $\rightarrow$ easier to find a suitable learning rate & learn more accurate models - scaling makes the loss function more symmetric & helps the gradient descent algorithm converge more quickly - stabilised convergence & accelerates training process - training process less sensitive to initial weights
Performing Feature Scaling: normalise separately for each input feature & once scaling values calculated on training set - can be applied to training/validation/testing sets
Gradient Checking: (has gradient descent been implemented correctly)
Calculate Weight Difference Before & After Gradient Descent:
$w^{(t)} = w^{(t-1)} - \alpha \cdot \frac{\partial L(w)}{\partial w} = \frac{w^{(t-1)} - w^{(t)}}{\alpha}$
Change the Weight by a Small Amount & Meaure Loss:
$\frac{\partial L(w)}{\partial w} = \lim_{\epsilon \rightarrow 0}\frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon} \approx \frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon}$

Unsupervised Learning

Cluster: a set of instances that are similar to each other & dissimilar to instances in other clusters
Similar: low intra-cluster variance (low variance between instances in the same cluster & high variance between instances not in the same cluster) $\rightarrow$ Implies that Data Points in each cluster are concentrated around the cluster's centroid (mean)
Clustering: task of grouping instances (in some feature space) such that instances in the same group are more similar to each other than instances in other groups
K-Means: dividing a set of features into K clusters such that each feature belongs to a cluster with the nearest mean (K $\rightarrow$ hyperparameter $\rightarrow$ no. clusters)
K-Means Algorithm:
Initialisation $\rightarrow$ select K & generate K random cluster centroids
Assignment $\rightarrow$ assign each training example to the nearest centroid
Update $\rightarrow$ update position of each centroid by computing the mean position of all examples assigned to it
Convergence Check: stop if position of centroids did not change (¡ threshold) - otherwise back to assignment
Elbow Method: run K-means with different Ks - keep track of loss for each K - select K with sharp shift in rate of decrease
Rationale: loss function improves as K increase - cannot choose K that minimises loss function (selects K - number of samples, minimising loss function at 0) - select K with improvement in loss function becoming less & less - not always optimal
Loss Function: sum of squared distances (from each pointer to cluster centroid/mean)
Limitations: elbow may not be very sharp & hard to determine exact cluster number - may not yield a clear answer for some datasets
Cross-Validation Method: dataset split into N folds - N-1 fold used as training dataset to compute centroids position - 1 fold used as validation dataset to compute average score for various K - best K selected - repeat for N possible splits & select best K on average (best K: such that further increasing K leads to only small performance improvement in average score)
Limitations: computationally expensive & much choose internal validation metric
K-Means Advantages: simple, easy to implement, & efficient (linear complexity)
K-Means Disadvantages: finds local optimum rather than global optimum (solution: run K-Means multiple times & choose model with lowest cost) - sensitive to initial centroid positions (solution: use a better initialisation) - only applicable if a distance function exists (for categorical data, use K-Mode with centroids representing most frequent values in cluster) - sensitive to outliers (outliers significantly shift position of centroid - solution: use algorithm that is less sensitive such a K-medoid using geometric median to determine centroids) - not suitable for clusters that are non a continuous & convex shape
Probability Density Estimation: PDE: estimating PDF p(x) from data
PDF p(x): function that models how likely a continuous variable is observed within a particular range
Applications: anomaly detection (compare PDR at a point to a threshold) - generative model (generate new samples from distribution) - discriminative model (directly model probability of observing output label given input sample)
Non-Parametric PDEs: (no assumption of underlying distribution - no. parameters can increase - high variance/low bias)
Histograms: groups data into contiguous bins - counting number of occurrences in each bin & normalising (area sums to 1)
$\rightarrow$ decreasing no. bins smooths PDF (decreasing variance & increasing bias)
Kernel Density Estimation: compute PDF by looking at training examples within a Kernel Function (Window)
Estimated PDF: kernel centered on each datapoint & summed up (normalised by volume of window & bandwidth)

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

N: total no. samples - h: bandwidth/window size - D: dimensions - H: Kernel Function
Normalising by Volume of Window ensures KDR integrates to 1 over the entire space (& valid PDF)
& Normalisation of Kernel by h ensures H does not depend on scale of data
Kernel Functions: H can take different functions - choice of H influences smoothness of resulting PDF (less critical than choice of h) (e.g. Gaussian Function gives a smooth approximation applied to window $\rightarrow$ PDR = sum of all bell curves)
Bandwidth: increasing bandwidth $\rightarrow$ smoother approximation (increases bias & decreases variance - more general estimate) - decreasing bandwidth $\rightarrow$ less smooth approximation (decreases bias & increasing variance - can lead to overfitting)
Parametric Approaches to PDE: (underlying assumption about data - fixed parameters - high bias & low variance)
Univariate Gaussian Distribution:
Compute PDF by fitting to Univariate Gaussian Mean & Variance to Training Examples (Larger Variance $\rightarrow$ Flatter & Spread Out)

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

Multivariate Gaussian Distribution: Compute PDF by fitting to Multivariate Gaussian Mean & Covariance Matrix to Training Examples

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

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N}x^{(i)}/; \hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \hat{\mu})(x^{(i)} - \hat{\mu})^T$$

Likelihood: reflects how well chosen parametric model fits the set of data (measuring probability of observing data x from dataset given set parameters) (a good model captures probability of generating/observing data within an interval)
Likelihood $= p(\mathbf{X}|\theta) = \prod_{i=1}^{N} p(x^{(i)}|\theta)$ (maximisation) (assumes training data independent & identically distributed)
Negative Log Likelihood $\mathcal{L} = -\log p(X|\theta) = -\sum_{i=1}^{N}\log p(x^{(i)}|\theta)$ (Minimisation)
$\rightarrow$ Fitting to Gaussian Distribution (as above) minimises the Negative Log Likelihood
Mixture Model: sum of weighted PDFs (reduces high variance of Gaussian Distributions & balances bias-variance trade off)

$$p(x) = \sum_{k=1}^{K}\pi_k p_k(x) \quad 0 \leq \pi_k \leq 1 \quad \sum_{k=1}^{K}\pi_k = 1$$

Gaussian Mixture Model (GMM): $p(x|\theta) = \sum_{k=1}^{K}\pi_k\mathcal{N}(x|\mu_k, \Sigma_k)$ (Weighted Mixture of Gaussians)
Expectation Maximisation: (Fitting GMMs to Training Examples by Iteratively Adjusting Parameters of the GMM to Maximise the Likelihood of Observed Data - cannot use Maximising Likelihood Directly as Updating Each Parameter depends on Other Parameters)
GMM-EM Algorithm: Initialisation: Select K (no. mixtures) & randomly initialise parameters
E-Step: Compute the Responsibilities for each Training Example & Each Mixture Component
M-Step: use responsibilities to update the parameters (mean/covariance/mixing ratio)
Convergence (stop if converged: if parameter not changed or negative log likelihood stagnated - otherwise back to E-Step)
Responsibility: probability that data point i was generate by component k (assigning weights to how much each Gaussian contributes to generation of each datapoint - if datapoint closer to mean of Gaussian - distribution considers to contribute more)

$$r_{ik} = \frac{\pi_k\mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x^{(i)}|\mu_j, \Sigma_j)} \quad \hat{\mu}_k = \frac{1}{N_k}\sum_{i=1}^{N}r_{ik}x^{(i)} \quad N_k = \sum_{i=1}^{N}r_{ik}$$

$$\Sigma_k = \frac{1}{N_k}\sum_{i=1}^{N}r_{ik}(x^{(i)} - \hat{\mu}_k)(x^{(i)} - \hat{\mu}_k)^T \quad \pi_k = \frac{N_k}{N} \quad \text{Converges to Local Optimum}$$

Selecting K: Minimise the Bayesian Information Criterion (BIC) (finds simplest model to fit data to accurate level)
BIC$_k = \mathcal{L}(K) + \frac{P_k}{2}\log(N)$
No. of Parameters: $P = (D \times K) + (D(D+1)/2) \times K + (K - 1)$ (Dimension D & K Components)
GMM-EM vs K-Means:
K-Means: (Hard Clustering) every point belongs to exactly one cluster - objective to minimise average mean squared distance - distance to the centroids is isotropic (all directions equal) - spherical clusters - parametric model - less computationally intensive (used when clusters assumed isotropic & well-separated
GMM: (Soft Clustering) every point belongs to several clusters with a certain probability - objective to minimise negative-log likelihood - each mixture component represents a different cluster & responsibilities define probability - can adapt clusters to different shapes - can generate clusters with different probabilities - assumes datapoints from Gaussian Distribution - more computationally intensive (used for more flexibility & when underlying assumption assumed Gaussian)

Evolutionary Algorithms (2)

Biased Roulette Wheel (Selection Operator): each individual given portion of circle proportional to fitness value & location random
$\rightarrow$ Compute Probability to Select Individual (Proportional to Fitness) - Compute Cumulative Probability (sum of selection probability of individual & all individuals before it) - Randomly Generate (Uniform Distribution) Number r between 0 & 1 (to randomly select individual) - Individual Selected whose Cumulative Probability Segment includes the Random Number r

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

Tournament (Selection Operator): Randomly Draw 2 Individuals from Population - Select Best out of the 2 - Repeat until Enough Parents
Elitism: Keep in the New Generation a Fraction of Best Individuals found so far (i.e. 10%) (best individual fitness cannot decrease)
Evolutionary Strategy: Genotype: List of Reals - Parent Selection: Uniform - Mutation: Gaussian
$\mu + \lambda - ES$: Randomly Generate a Population of $(\mu + \lambda)$ Individuals - Evaluate Population - Select Best μ Individuals as Parents (x) - Generate λ Offspring (y) from Parents - Population: Union of Parents & Offspring - Return to Evaluation

$$y_i = x_j + \mathcal{N}(0, \sigma) \quad \text{pop} = (\bigcup_j^{\lambda} x_j) \cup (\bigcup_j^{\mu} y_j)$$

Adapt Sigma over Time: $x'_j = \{x_j, \sigma_j\}q; \sigma_i = \sigma_j\exp(\tau_0\mathcal{N}(0, 1)) \ y_i = x_j + \sigma_i\mathcal{N}(0, 1)$
τ_0 : Learning Rate & $\tau_0 \propto \frac{1}{\sqrt{n}}$ (n: dimension of genotype) (Typically $\mu/\lambda = 5$)