COMP70053: Introduction to Machine Learning Notes

1 The Big Picture

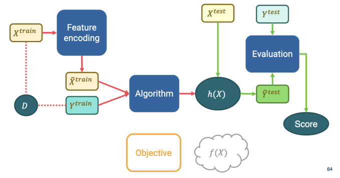
- AI aims to act and think both *humanly* and *rationally*.
- A computer program learns from experience `E` in some class of tasks `T` with performance measure `P`, if its performance (measured by `P` on tasks in `T`) improves with experience `E`.
- Three main settings:
  1. **Supervised learning**
     - input variables have correct output labels attached.
  2. **Unsupervised learning**
     - input variables have no labels attached.
     - aim to discover hidden/latent structures within the data.
     - e.g. clustering, dimensionality reduction.
  3. **Reinforcement Learning**
     - input variables have no labels attached, but the environment returns a reward signal for each action.
     - **policy search**: finding which action will maximise reward depending on the agent's current state.
     - e.g. video game AI, robotics.
- Other settings:
  1. Semi-supervised learning
     - some data have labels, some do not
  2. Weakly-supervised learning
     - inexact output labels
     - e.g. there is an umbrella in the photo, find it
- Two most popular ML tasks:

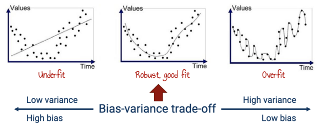
	Classification	**Regression**
Output	discrete/categorical	real-valued/continuous
Used for	classifying data under one or more labels	predicting a quantity related to the data
Variants	binary, multi-class, multi-label	simple, multiple, multi-variate

- Two kinds of ML algorithms

Lazy Learner	**Eager Learner**
Stores the training examples and postpones Fising beyond these data until an explicit request is made at test time.	Constructs a general, explicit description of the target function based on the provided training examples.

- The supervised learning pipeline: The bias-variance trade-off:





- **Feature encoding**: convert raw feature values to machine-friendly format (typically normalised between values: 0-1)
- **Curse of dimensionality**: higher dimension data leads to: *increased computational complexity, data sparsity, overfitting.*

2 K-Nearest Neighbours and Decision Trees

- **Nearest Neighbour classifier**: classify a test instance to the class label of the nearest training instance (acc to some distance metric).
- **k-Nearest Neighbours (kNN) classifier**
  - classify based on the class label with the greatest *weighted average* amongst the k nearest neighbours.
  - k is usually an odd number to avoid equally weighted labels.
  - k must be chosen appropriately with a validation set.

Choice of k	**Too Low**	**Too High**
Noise sensitivity	too high	too low
Quality of fit	will overfit	will underfit

  - (**distance weighted**) assign weights $w^{(i)}$ to each neighbour based on proximity and choose the class with the largest weighted sum.
  - simple, powerful, but slow for large datasets (curse of dimensionality)
- **Decision Tree Learning**: a method for approximating discrete classification functions by means of a tree-based representation.
  - General algorithm:
    1. Search for an 'optimal' splitting rule on training data.
    2. Split dataset according to chosen splitting rule.
    3. Repeat 1 and 2 on each new split subset.
- **Entropy**: a measure of the *uncertainty* of a random variable; the *average* amount of information:
$$H(X) = -\sum_k^K P(x_k)\log_2(P(x_k))$$
- **Information gain**: the difference between the initial entropy and the weighted average entropy of the produced subsets.
$$IG(D, subsets) = H(D) - \sum_{S \in subsets} \frac{|S|}{|dataset|} H(S)$$
- **Types of inputs**

	Ordered	**Categorical**
Split	compare value with number (e.g. $X < 10$)	split based on all possible labels of a feature
Tree	binary	multiway
Process	for each feature, sort its values and consider split points between two examples with different class labels	search for most informative feature and split based off that
Other	can split on a single feature more than once	guaranteed to split on a single feature at most once

- Dealing with **overfitting**:
  - Early stopping (with *max tree depth* or *min examples per leaf*)
  - Pruning (with a validation set)
- The **pruning** process:
  1. Go through each node only connected to leaf nodes.
  2. Turn each into a leaf node (with majority class label).
  3. Evaluate pruned tree on validation set.
  4. Keep tree pruned if accuracy is higher else, revert.
  5. Repeat until all nodes have been tested.
- **Random Forests**: model with many decision trees voting on the class label; each tree is trained on a random sample of the training set and a random subset of features.

3 Machine Learning Evaluation

- **Hyperparameters (HP)**: model parameters chosen before training.
- **Hyperparameter tuning**
  - find HP values that lead to the best performance.
  - split dataset into: training/validation/test (usually 60%/20%/20%)
  - try different HP values on *training set*, choose one with best accuracy on *validation set*, and perform final evaluation on *test set.*
- General rules to follow
  - Evaluate the model on a held-out (test) dataset.
  - The test dataset should *not* be used to train nor validate the model.
  - Assume labels of the test set are only provided after training.
- **Cross-Validation (CV)**
  - procedure:
    1. divide the dataset into k (usually 10) equal folds; use $k-1$ for training/validation and one for testing.
    2. iterate k times, each time with a different test set
    3. evaluate average performance across the folds
  - for parameter tuning, either:
    * Run CV using the validation set to select the best parameters, then choose the model with the best test set performance.
    * Run CV, where, at each fold, we run an internal CV across the $k-1$ folds to find optimal HPs, then choose the model with the best test set performance overall.
  - useful when the available dataset is small in size.
- **Confusion matrix**

	Class 1 Predicted	**Class 2 Predicted**
Class 1 Actual	TP: True Positive	FN: False Negative
Class 2 Actual	FP: False Positive	TN: True Negative

- **Precision** $= P(\text{positive} \mid \text{classified positive}) = \frac{TP}{TP+FP}$
- **Recall** $= P(\text{classified positive} \mid \text{positive}) = \frac{TP}{TP+FN}$
- one might be preferred over another depending on the application:
  - high *recall*, low *precision*: most of the positives are recognized, but there are many false positives.
  - low *recall*, high *precision*: miss most positives, but those classified are truly positive.
- $F_1 = \frac{2}{\frac{1}{precision}+\frac{1}{recall}}$, $F_\beta = \frac{1+\beta^2}{\frac{\beta^2}{precision}+\frac{1}{recall}}$
- note: *precision*, *recall*, and *F1* are computed for each class separately.
- class-wide performance can be measured through:
  - **Macro-averaging**: take average on the class level, i.e. find the average of each class' metrics.
  - **Micro-averaging**: take average on item level, i.e. sum up TP, FP, TN, FN from all classes, and calculate metrics altogether.
- If test set is imbalanced, either: **downsample** the majority class (choose a subset) or **upsample** the minority class (add duplicates).
- accuracy is misleadingly swayed by the majority class; macro-averaged recall knows nothing about FPs; F1 is affected by class imbalance.
- **Confidence interval for error**: $err_s \pm z_\alpha \sqrt{\frac{err_s \cdot (1-err_s)}{n}}$.

α	50	68	80	90	95	98	99
z_α	0.67	1.00	1.28	1.64	1.96	2.33	2.58

- **P-hacking**: misuse of data analysis to find statistically significant patterns when in truth no underlying effect exists.

4 Neural Networks I

- **Linear regression**
  - finding the linear trend-line (and its corresponding parameters) that best describe data points across the feature space, e.g. $\hat{y} = ax + b$.
  - loss function: $E = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$
  - **Gradient descent**: repeatedly update parameters a and b by taking small steps in the negative direction of the partial derivative.
    $$a := a - \alpha\frac{\partial E}{\partial a} = a - \alpha\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})x^{(i)}$$
    $$b := b - \alpha\frac{\partial E}{\partial b} = b - \alpha\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})$$
  - as it is analytically solvable, no iterated solutions are necessary: $\theta^* = (X^TX)^{-1}X^Ty$ for $X = [x^{(i)} \mid 1.0]$, $y = [y^{(i)}]$, $\theta = [a, b]^T$; though not great for large problems (matrix inversion is $O(n^3)$).
- **Artificial neuron**
  - inspired by biological neurons which receive and release signals.
  - given inputs $x = [x_i]$, weights $W = [\theta_i]$, and activation function g, it returns an output of $\hat{y} = g(W^Tx)$.
  - can be extended to a **multilayer perceptron (MLP)** where neurons are connected in sequence to learn higher-order features.
- **Perceptron**
  - an early version of an artificial neuron used for supervised binary classification, particularly for *linearly separable* functions.
  - uses **threshold function** as an activation: $h(x) = \begin{cases} 1 & \text{if } W^Tx > 0 \\ 0 & \text{otherwise} \end{cases}$ with learning rule: $\theta_i \leftarrow \theta_i + \alpha(y - h(x))x_i$.
- **Activation functions**

	Linear	**ReLU**	**Sigmoid**	**Tanh**
$g(x)$	x	$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$\frac{1}{1+e^{-x}}$	$\tanh(x)$
$g'(x)$	1	$\begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$	$g(x)(1-g(x))$	$1 - g(x)^2$
bounds	$(-\infty, \infty)$	$[0, \infty)$	$[0, 1]$	$[-1, 1]$

5 Neural Networks II

- **Loss function**
  - useful for *gradient descent* if differentiable: $\theta_i^{t+1} = \theta_i^t - \alpha \cdot \partial E/\partial\theta_i^t$.
  - **mean squared error** for regression: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$.
  - **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 $C :=$ set of classes and $\hat{y}_c^{(i)} :=$ predicted probability of class c for datapoint i.
- **Backpropagation**
  - an iterative calculation for partial derivatives (gradients) used to update (and thus optimise) the weights of a neural network.
  - the gradient for a node's weight can be expressed by gradients of those that come after it; these are iteratively fed to nodes backwards.
  - given $\frac{\partial Loss}{\partial Z}$, we can update weights and biases with:
    $$\frac{\partial Loss}{\partial W} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial W} = X^T\frac{\partial Loss}{\partial Z}$$
    $$\frac{\partial Loss}{\partial b} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial b} = \mathbf{1}^T\frac{\partial Loss}{\partial Z}$$
    and pass $\frac{\partial Loss}{\partial X} = \frac{\partial Loss}{\partial Z} \cdot \frac{\partial Z}{\partial X}$ to the lower layer.
  - given $\frac{\partial Loss}{\partial A}$, we can find $\frac{\partial Loss}{\partial Z}$ with:
    $$\frac{\partial Loss}{\partial Z} = \frac{\partial Loss}{\partial A} \circ g'(Z) = \begin{bmatrix} \frac{\partial Loss}{\partial a_{1,1}}g'(z_{1,1}) & \frac{\partial Loss}{\partial a_{1,2}}g'(z_{1,2}) \\ \frac{\partial Loss}{\partial a_{2,1}}g'(z_{2,1}) & \frac{\partial Loss}{\partial a_{2,2}}g'(z_{2,2}) \end{bmatrix}$$
- **Gradient descent**
  - initialise weights randomly, compute gradient based on *whole data set*, and update weights; data sets are often too big for this.
  - (**stochastic**) loop over each *data point*: compute gradient based on a data point and update weights; often noisy for single data point.
  - (**mini-batched**) loop over *batches of data points*: compute gradient based on a batch and update weights; widely used in practice.
- Ways to **optimise** neural networks.
  - tune, add a decay, or use an adaptive learning rate
  - initialise weights differently (zeros, normal, xavier glorot/uniform)
  - normalise data (to make weight updates proportional to the input)
    - **min-max normalisation**: $X' = a + \frac{(X - X_{min})(b-a)}{X_{max} - X_{min}}$
    - **standardisation**: $X' = \frac{X - \mu}{\sigma}$
- Finding the **best fit**
  - increase/reduce the network's **capacity** - the number of neurons, layers, or parameters - if the network is underfitting/overfitting.
  - use a validation set to **stop training early**.
  - (**regularisation**) add constraints to prevent model from overfitting.
  - (**dropout**) randomly set neural activations to zero during training

6 Unsupervised Learning

- **Clustering**: grouping instances (in some feature space) such that those in the same group are more similar than those in other groups.
- **K-Means**
  1. **initialisation**: randomly select K training examples as centroids
  2. **assignment**: assign each training example to the nearest centroid, i.e. for each $i \in \{1, \ldots, N\}$
     $$c^{(i)} = \min_{k\in\{1,\ldots,K\}} \|x^{(I)-\mu_k}\|^2$$
  3. **update**: update position of each centroid to mean position of examples assigned to it, i.e. for each $k \in 1, \ldots, K$
     $$\mu_k = \frac{\sum_{i=1}^{N}\mathbf{I}(c^{(i)}=k)\cdot x^{(i)}}{\sum_{i=1}^{N}\mathbf{I}(c^{(i)}=k)}$$
  4. **convergence check**: stop if the position of centroids barely changed, i.e. if $\forall k|\mu_k^{(t)} - \mu_k^{(t-1)}| < \epsilon$, else go to step two.
- An appropriate **choice** for K
  - **elbow method**: run K-means with different Ks; keep track of loss for each K; select K where rate of decrease sharply shifts.
  - **CV method**: choose K with the best average CV performance.
- K-means is simple, popular, and efficient: $O(\text{iters} \cdot \text{clusters} \cdot \text{examples})$. However, a poor choice of K leads to local optimums sensitive to initial centroid positions, and it requires a distance function, is sensitive to outliers, and is not suitable for non-hyper-ellipsoid clusters.
- **Probability density estimation**
  - estimates the true PDF by which the data is distributed.
  - **non-parametric methods** (no assumptions about form) include *histograms* and *kernel density estimators.*
  - **parametric methods** (makes simplified assumptions about form).
- **Gaussian distribution**
  - assume data is normally distributed and find MLEs $\hat{\mu}$ and $\hat{\sigma}^2$.
  - **univariate**
    - $\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp(-\frac{(x-\mu)^2}{2\sigma^2})$.
    - $\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N}x^{(i)}$, $\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \hat{\mu})^2$.
  - **multivariate**
    - $\mathcal{N}(\mathbf{x} \mid \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D|\Sigma|}} \cdot \exp(-\frac{1}{2}(\mathbf{x} - \mu)^T\Sigma^{-1}(\mathbf{x} - \mu))$.
    - $\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N}x^{(i)}$, $\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}^{(i)} - \hat{\mu})(\mathbf{x}^{(i)} - \hat{\mu})^T$.
- **Gaussian mixture models (GMM)**
  - a weighted mixture of Gaussians; a "soft K-means clustering".
  - optimised using **Expectation-Maximisation (EM)**:
    1. **initialisation**: randomly initialise parameters
    2. **E-step**: compute responsibilities $r_{ik} = \frac{\pi_k\mathcal{N}(\mathbf{x}^{(i)}|\mu_k,\Sigma_k)}{\sum_j^K \pi_j\mathcal{N}(\mathbf{x}^{(i)}|\mu_j,\Sigma_j)}$.
    3. **M-step**: update *mean* $\hat{\mu}_k = (\sum_{i=1}^{N} r_{ik})^{-1}\sum_{i=1}^{N} r_{ik}\mathbf{x}^{(i)}$.
    4. **convergence check**: check for changes or likelihood stagnation.

7 Evolutionary Algorithms

- **Evolutionary algorithms (EA)**
  - an optimisation algorithm for *black-box functions* (no gradient).
    1. maintain a (randomly generated) **population** of solutions.
    2. evaluate their *phenotypes* with a **fitness function**.
    3. rank and select the **fittest** to start a new population.
  - stop when: a specific fitness value reached, a pre-defined number of generations reached, or the best fitness in the population stagnates.
- governed by three main **operators**:
  - **selection**
    - choose individuals to be parents in the next generation.
    - (**biased roulette wheel**) individuals have some probability p_i of being chosen from the population.
    - (**tournament**) two individuals are randomly selected, and the better of the two is chosen; repeat until there are enough parents.
    - (**elitism**) keep a fraction (10%) of the best individuals for the new generation; guarantees fitness does not decrease per generation.
  - **cross-over**
    - combine the genotypes of the parents.
    - (**single-point**) a split point is randomly picked, and the offspring is formed by exchanging portions of the genotype.
  - **mutation**
    - apply variations to solutions; explore nearby solutions.
    - (**standard** on binary strings) each bit is flipped with some probability m, often fixed to $1/|\text{genotype}|$.
- **$(\mu + \lambda)$ - ES**
  1. randomly generate a population of $(\mu + \lambda)$ individuals.
  2. (evaluate and) select the best μ individuals as parents (x).
  3. generate λ offsprings $y_i = x_j + \mathcal{N}(0, \sigma)$, with $j = randi(\mu)$.
  4. new population becomes the union of parents and offspring; go to (2).
- $(\mu + \lambda)$ σ: too large: population moves quickly to solution but hard to refine; too small: population moves slowly and prone to local optima.
- **Novelty search (NS)**
  - optimise novelty rather than the quality of a solution; replace fitness evaluation with a search through the novelty archive.
  - **behavioural descriptor (BD)** characterises aspects of solutions.
- **Quality diversity (QD)**
  - aim to find collection of **diverse** and **performant** solutions, using both a **behavioural descriptor** and a **fitness function**.
  - **general framework**: stochastic selection, random mutation, evaluation, tentative addition to collection (either in grid or unstructured).
  - **NS with local competition**: archive those outperforming kNN.
- **MAP-elites**
  - discretise BD space in a grid and fill with best solutions.
  - solutions go to cells corresponding to their BD; if the cell is empty, it is added; if the cell is occupied, keep one with the best fitness.
  - **diversity** quantified with *archive size*, **performance** with max/mean fitness; **convergence speed** is also of importantance.
  - **QD-score**: sum of the fitness of all solutions in the archive.