Uninformed Searches

BFS: level by level expansion, complete: yes, optimal: yes (if uniform step costs), time/space: $O(b^d)$

DFS: depth first, complete: no (if infinite search space), optimal: no, time: $O(b^m)$, space: $O(bm)$

UCS: expands the least cost node first (lowest $g(n)$), complete: yes (if step costs are positive), optimal: yes, time: $O(b^{\lfloor C*/\epsilon \rfloor})$, similar to BFS, but we sort the fringe by $g(n)$.

DLS: DFS with a depth limit l, complete: no, optimal: no, time: $O(b^l)$, space: $O(bl)$.

IDS: DLS with increasing depth limits, complete: yes (if finite search space), optimal: yes (uniform step costs), time: $O(b^d)$, space: $O(bd)$,

Informed Searches

Greedy Search: expands the node with the lowest $h(n)$. complete: yes (if m is finite). optimal: no. time/space: $O(b^m)$.

A* search: expands the node with the lowest $f(n)$, where $f(n) = g(n) + h(n)$ **admissable heuristic**: $h(n) \leqslant h^*(n)$ the true cost to the goal complete/optimal: if $h(n)$ is admissible if $f(n) = g(n)$ A* = UCS if $f(n) = depth(n)$ A* = BFS

if $h(n)$ is **consistent**, then $h(n) \leqslant g(n') + h(n')$ for all nodes n' that are successors of n, consistent heuristics are also admissible.

dominant heuristic: $h_1(n) \geqslant h_2(n)$ for all nodes n, then h_1 is dominant over h_2.

Local Search Algorithms

Hill Climbing: expands the node with the best $v(n)$, but only considers successors of the current node. solution found depends on the initial node. complete/optimal: no (local optimum, plateaus, ridges)

Beam Search: keeps the k best nodes (best $v(n)$) at each level. complete/optimal: no

Simulated Annealing: randomly selects successors if the successor is better than the current node, it is selected. if $v(m)$ is better than $v(n)$, then m is selected with probability else $n = m$ with probability p where

$p = e^{-\frac{v(m)-v(n)}{T}}$

bad moves are $v(m) > v(n)$ if looking for min

T decreases over time, starting from a high value.

Thereom: if T is high enough/lowers slowly enough, then the algorithm will eventually find the global optimum. i.e. complete/optimal

Genetic Algorithms: select best individuals, crossover, mutate best individuals are selected based on a fitness function.

crossover is combining swapping parts of two individuals about a point to create a new individual.

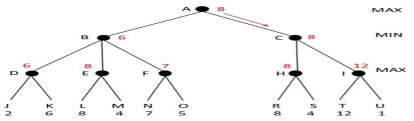
mutation is random change of bits to an individual.

complete/optimal: no

Games

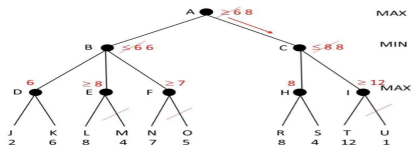
deterministic vs chance, e.g. dice vs no dice games perfect vs imperfect information: e.g. chess vs poker zero-sum vs non-zero-sum: e.g. one's gain is another's loss vs both can win process forward, not from goal, too many possible goals, don't learn anything if incomplete search from goal

Minimax: used for two-player games, where one player tries to maximize their score and the other tries to minimize it. perform DFS to the terminal node of the game tree, then backtrack to the root. At max: pick the maximum value of the children, at min: pick the minimum value of the children. complete/optimal: yes same time/space complexity as DFS, $O(b^d)$



Alpha-Beta Pruning: optimizes the minimax algorithm by pruning branches that cannot possibly influence the final decision. traverse tree in DFS order apply utility function to terminal nodes compute values backwards

At each non-leaf node store the value indicating the best move for the player to play. basically pick the best move for the current player, if max choose the maximum value of the children, if min choose the minimum value of the children. if the value of a child is worse than a parent, prune it



Imperfect minmax and alpha-beta pruning takes too long for large trees, so we cut off the search at a certain depth, and evaluate the nodes using a heuristic function. however with cutoff depth we may cutoff before a losing move so we do a secondary search past the cutoff point to ensure that there are no hidden pitfalls. i.e. we do not want to stop at non-quiescent moves, moves that lead to large change in adavantage.

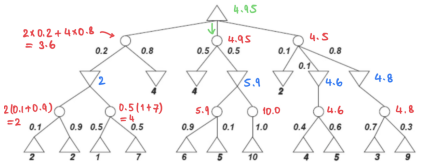
Expectiminimax: used for games with chance, where the outcome of a move is not deterministic.

value of the parent node is the expected value of the children nodes, weighted by the probability of each child node.

i.e.

$$v(n) = \sum_{i=1}^{k} p_i v(n_i)$$

where p_i is the probability of child node n_i and $v(n_i)$ is the value of child node n_i.



Supervised Learning

Nearest Neighbour example of a lazy classifier, it stores all training examples but does not build classifier until unlabled example is given. commonly used distance metrics are:

euclidean:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

manhattan:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

minkowski:

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

need to normalise the data before using distance metrics, otherwise features with larger ranges will dominate the distance metric.

k-NN: majority vote upon the k nearest neighbours.

very sensitive to k, generally we use $k = \sqrt{n}$ where n is the number of training examples.

can be used for classification and regression.

distance for nominal features is 0 if the feature is the same, 1 otherwise.

for missing values/different features the distance is 1, if the two attributes are the same and not missing then 0.

NN is good with lower dimensions, but suffers from the **curse of dimensionality**. as the number of dimensions increases, the distance between points becomes less meaningful, and the nearest neighbours may not be representative of the data. so we can select a subset of features to reduce the dimensionality of the data. sensitive to noise makes predictions based on the local structure of the data, so it is a local model, not a global model.

1-Rule a simple classifier that uses only one feature to classify the data. it finds the feature that gives the best classification accuracy, i.e. fewest errors and uses that feature to classify the data. missing values are treated as a separate value, nominal features are decretized. this may lead to overfitting, so we impose a minimum number of examples per rule. and merge adjacent rules with the same class.

Naive Bayes probabilistic classifier based on Bayes' theorem.

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

to calculate $P(E|H)$ we use the conditional independence assumption,

$$P(E|H) = \prod_{i=1}^{n} P(E_i|H)$$

where E_i are the features of the evidence, and n is the number of features.

where H is the hypothesis and E is the evidence. assumes that each feature is equally important, and independent of each other.

Laplace correction is used to avoid zero probabilities, add 1 to the numerator and k to the denominator, where k is the number of possible values for the feature.

$$P(E_i|H) = \frac{N_{E_i,H} + 1}{N_H + k}$$

where $N_{E_i,H}$ is the number of examples with feature E_i and class H,

Missing attribute values are ignored. e.g. outlook=?, temperature=cool, humidity=high, windy=true ignore outlook, use the rest to classify.

Handling numeric attributes: we can use PDF to model the distribution of the data.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean and σ is the standard deviation of the feature.

advantages: simple, clear, fast $O(pk)$ p = # of training examples, k = # of attributes. robust to noise, works well with small datasets.

correlation reduces the performance of Naive Bayes, violates the independence. so disregard correlated features.

numeric attributes aren't always normally distributed, so we can discretize to nominal Use alternative PDFs (Poisson, binomial, gamma) Apply normalizing transformations Kernel density estimation

Evaluating Classifiers

holdout validation: split the data into training and test sets, train on the training set, and evaluate on the test set.

accuracy = 1 − Error rate

split the data into training, validation, and test sets, train on the training set, evaluate on the validation set, and test on the test set. holdout validation becomes more reliable by repeating the process multiple times and averaging the results, this is called **cross-validation**.

Stratifcation ensure that each class is represented in the training and test sets, i.e. the proportion of each class in the training and test sets is the same as in the original dataset.

Comparing Classifiers calculate the difference in accuracy between the classifiers, calculate the standard deviation of the difference, calculate the confidence interval for the difference, if the confidence interval include 0 insignifcant, else significant.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{k}(d_i - d_{mean})^2}{k-1}}$$

$$Z = d_{mean} \pm t_{(1-\alpha)(k-1)} \frac{\sigma}{\sqrt{k}}$$

Z is the confidence interval for the difference in accuracy, generally we use a 95% confidence level, k is number of folds, t is obtained from the t-distribution table, $(1 - a)$ is the confidence level, $(k - 1)$ is the degrees of freedom.

Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

$$P = \frac{TP}{TP + FP} \quad R = \frac{TP}{TP + FN}$$

$$F_1 = \frac{2PR}{P + R}$$

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

(P): proportion of true positives among all predicted positives. (left col accuracy)

(R): proportion of true positives among all actual positives. (top row accuracy)

(F1): harmonic mean of precision and recall, used to balance the two metrics.

Inductive learning: produce a general rule from specific examples

Decision Trees

topdown approach, recursively split the data into subsets based on the features, until a stopping criterion is met.

Select the feature that maximizes the information gain, i.e. the feature that best separates the data into subsets.

Split the data into subsets based on the selected feature, and repeat the process for each subset.

recurse on each subset

Stop when all examples in the subset have the same class. No further splits are possible (e.g. no features left, or all features are the same).

each root to leaf path represents a rule.

i.e. $(x \wedge y)$

and the entire tree is a conjunction of the rules.

i.e. $(X_1 \vee X_2 \vee X_3 \vee \ldots \vee X_i)$

Entropy is a measure of the impurity of a set of examples,

$$H(S) = -\sum_{i=1}^{c} P_i \log_2(P_i)$$

where p_i is the proportion of examples in class i in the set S, and c is the number of classes.

Information gain is the reduction in entropy after splitting the data on a feature.

$$IG(S|A) = H(S) - \sum_{j \in values(A)} \frac{|S_v|}{|S|} H(S_v)$$

where $H(S)$ is the entropy of the set S, S_v is the subset of examples in S that have value v for feature A, and $values(A)$ is the set of all possible values for feature A.

basically it's entropy of the class - proportion of examples in each feature * entropy of the example in the feature.

DT Pruning:

pre-pruning: stop growing the tree before it is fully grown.

post-pruning: grow the tree fully then prune.

post-pruning is more successful: sub-tree replacment: replace subtree with majority class of the subtree's examples. sub-tree raising: raise a subtree to the parent node if it is more accurate than the parent node. i.e. if the subtree has more examples than the parent.

numeric attributes are handled by splitting the data into ranges, e.g. $x < 5$ or $x \geqslant 5$.

Neural Networks

neurons are connected in layers, each neuron has a weight, and each neuron has bias. weights and biases are intialised randomly. neuron perform some step computation

$$a_i = f(\mathbf{w_i} \cdot \mathbf{p} + b_i)$$

Perceptron is a single layer neural network. uses the step function:

$$a = f(\mathbf{w} \cdot \mathbf{p} + b)$$

where $f(x) = 1$ if $x \geqslant 0$, else $f(x) = 0$. the perceptron can only learn linearly separable functions. it is a binary classifier, i.e. it can only classify examples into two classes.

Perceptrons learn by: - initialising weights and biases randomly - feeding examples into the perceptron one at a time until an epoch is classified correctly.

$e = t - a$

$w_{new} = w_{old} + ep^T$

$b_{new} = b_{old} + e$

where e is the error, t is the true label, a is the predicted label, and p is the input vector.

Multi-Layer Perceptron instead of a single layer we have multiple layers of neurons, neurons only recieve input from previous layers, all neurons in a layer are connected to all neurons in the next layer. layer size is average of the input and output layer size. use hot vector encoding for the output layer, i.e. if there are k classes, then the output layer has k neurons, each neuron represents a class.

sigmoid: $f(x) = \frac{1}{1+e^{-x}}$

$f'(x) = f(x)(1 - f(x))$

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

$f'(x) = 1 - f(x)^2$

Backpropagation

backpropagation is used to train multi-layer perceptrons, it is a supervised learning algorithm that uses gradient descent to minimize the error of the network. the error is calculated as the difference between the true label and the predicted label.

each non-input neuron computes

$net_j = \sum_{i=1}^{m} w_{ij} a_i + b_j \quad o_j = f(net_j)$

where f is a differentiable activation.

output layer error: $\delta_j = f'(net_j)(d_j - o_j)$

hidden layer error: $\delta_j = f'(net_j) \sum_i w_{ji} \delta_i$

weight change: $\Delta w_{ij} = \eta \delta_p a_p$

$E = \frac{1}{2} \sum_{i=1}^{n} (d_i - a_i)^2$

backprogation algorithm: init: random weight and bias repeat: forward pass: calculate the output of the network backward pass: calculate the error of the output layer and propagate it back to the hidden layers update weights and biases using the error and the learning rate η. until the error falls below threshold.

Deep Learning

deep learning have more than one hidden layer, automatically learns features from the data. we use multiple layers because they can learn more complex features than a single layer.

Autoencoders NN: used for pre-training, dimensionality reduction, compression, encryption. we train first autoencoder to learn a compressed representation of the data, then we train a second autoencoder to learn a compressed representation of the first autoencoder's output. after pretraining, we add a supervised layer on top of the autoencoders, and train the entire network using backpropagation. other types of au-

toencoders include: sparse - more hidden neurons than input neurons, denoising - add noise so NN learns robust features.

Convolutional NN: process raw pixels, handle distortions, and learn heirarchical features. each neuron connected to a small region of the input image, called a **receptive field**. A single filter (**kernel**) is applied to the input image which reduces the dimensionality of the input image. **pooling** is used to reduce the dimensionality of the output of the convolutional layer, e.g. max pooling, taking the maximum value of a region of the output image. **LCN** is used to normalise the output of the of pooling layer, it normalises the output by subtracting the mean and dividing by the standard deviation of the output.

$$\text{LCN}(x) = \frac{x - \mu}{\sigma}$$

LCN allows for brightness invariance.

SVM

in linear case SVM finds the hyperplane with the maximum margin that separates the data into two classes. the margin is the distance between the hyperplane and the closest data points from each class, called support vectors.

$w^\top x + b = 0$

that separates the data into two classes,

$sign(w^\top x + b)$ is the class label.

SVM chooses the hyperplane that maximises the margin. minimising: $\frac{1}{2}||w||^2$

subject to the linear constraint:

$y_i(w^\top x_i + b) \geqslant 1$

The optimization problem can be solved using Lagrange multipliers, leading to the dual problem:

$$\max = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{n} \lambda_i \lambda_j y_i y_j x_i \cdot x_j$$

optimal decision boundary is given by:

$$w = \sum_{i=1}^{n} \lambda_i y_i x_i$$

where λ_i are the Lagrange multipliers, y_i are the class labels, and x_i are the support vectors.

we can allow misclassification when problem is not linearly separable, add an addtional parameter C to the optimization problem to maximise trade-off between margin and misclassification.

usually we use the kernel trick to map the data into a higher-dimensional space, where it is linearly separable. The optimization problem becomes:

$$\max = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{n} \lambda_i \lambda_j y_i y_j K(x_i \cdot x_j)$$

$$w = \Phi(x) = \sum_{i=1}^{n} \lambda_i y_i \Phi(x_i)$$

where $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$ is the kernel function, which maps the input data into a higher-dimensional space.

common kernel functions include:

polynomial kernel: $K(x_i, x_j) = (x_i \cdot x_j + c)^d$

RBF kernel: $K(x_i, x_j) = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}}$

tanh kernel: $K(x_i, x_j) = \tanh(\alpha(x_i \cdot x_j) + c)$

classification becomes

$$f(x) = sign\left(\sum_{i=1}^{n} \lambda_i y_i K(x_i, x) + b\right)$$

Ensemble

Bagging: bootstrap aggregating, splits the training data into subsets by sampling with replacement, trains a classifier on each subset, and combines the predictions of the classifiers by averaging or voting.

Boosting: train a model in sequence, increase weights of misclassfied and decrease weights. future models will use new weights to classfy the data. in final ensemble, each model is weighted by its accuracy.

AdaBoost: if the learning algo is weak, i.e. accuracy is less than 50%, the adaboost will increase the weights of the misclassified examples, and decrease the weights of the correctly classified examples.

Random Forest: an ensemble of decision trees, get training data by bagging. each tree is trained on a random subset of the features, and the final prediction is made by averaging or voting the predictions of the trees. tree is made how a decision tree is made. no pruning.

Bayesian Networks

probability chain rule:

$P(A, B, C) = P(A|B, C)P(B|C)P(C)$

these are used to represent the probabilistic relationships between variables.

computing join probability BN

$$P(x_1, ..., x_n) = \prod P(x_i | Parents(i))$$

Full-Joint Enumeration: the sum over all hidden variables $P(X|e) = \alpha \sum_{Y,Z,\dots} P(X, e, Y, Z, \dots)$

Variable elimination: factorise the joint into conditional probability table (CPT) factors, eliminate hidden vars one at a time by multiplying relevant factors, summing out the variable to produce a new factor. Multiply remaining factors and normalise.

Diagrams: note chance = oval, decision = square, utility = diamond. To solve, roll back network starting at leaves, then computing expected utility at each decision node given the parents. Then, select the action maximising EU:

$$EU(a) = \sum_s P(s|parents) \cdot U(s, a)$$

For node Storm (S):

s	~s
0.31	0.69

For node Cat (C):

	c	~c
s	0.75	0.25
~s	0.22	0.78

For node Alarm (A):

		a	~a
b	c	1	0
b	~c	0.67	0.33
~b	c	0.25	0.75
~b	~c	0.2	0.8

For node Burglar (B):

	b	~b
s	0.25	0.75
~s	0.33	0.67

We need to compute P(A=truel S=true).

$$P(a|s) = \frac{P(a,s)}{P(s)} = \frac{\sum_{b,c} P(a,s,B_p,C_c)}{P(s)}$$

$P(a|s) = \sum_{b,c} P(a,s,B_p,C_c) =$

$= \sum_{b,c} P(a|B_p,C_c)P(s)P(B_p|s)P(C_c|s) =$

$= P(s)\sum_b P(B_p|s)\sum_c P(C_c|s)P(a|B_p,C_c) =$ <- Pushing the summation over c in:

$= 0.31 * \sum_b P(B_p|s)[0.75 * P(a|B_p,c) + 0.25 * P(a|B_p,\neg c)] =$ <- Summation for c and ~c

$= 0.31 * [0.25 * (0.75 * 1 + 0.25 * 0.67) + 0.75 * (0.75 * 0.25 + 0.25 * 0.2)] =$ <-Summation for b and ~b

$= 0.31 * (0.25 * 0.9175 + 0.75 * 0.2375) =$

$= 0.31 * 0.4075 =$

$= 0.1263$

$P(a|s) = \frac{P(a,s)}{P(s)} = \frac{0.1263}{0.31} = 0.4074$

Clustering/Unsupervised Learning

centroid is mean of cluster. metroid is median of cluster.

single links: smallest link between elements of two clusters **complete links**: largest **average links**: average

good clustering has high intra-cluster similarity, low inter-cluster similarity.

measured with David-Bouldin index,

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j,j \neq i} \frac{dist(\mathbf{x}, c_i) + dist(\mathbf{x}, c_j)}{d_{c_i,c_j}}$$

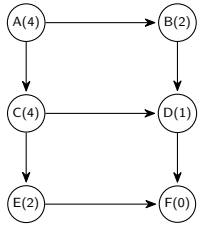
k is number of clusters, c_i is the centroid of cluster i, $dist(\mathbf{x}, c_i)$ is average distance between points and its cluster's centroid, $dist_{c_i,c_j}$ is distance between centroids of clusters i and j.

K-Means Clustering: we have k centroids. assign each point to their closest centroid, then recalculate the centroids as the mean of the points assigned to them. repeat until the centroids do not change or a maximum number of iterations is reached.

Nearest Neighbour Clustering: start with one point as a cluster, the for each new point merge it with an existing cluster if the distance between the point and the cluster is over a threshold t. start a new cluster.

Hierarchical Clustering: builds a denogram, a tree-like structure that represents the clustering of the data. agglomerative: start with each point as a cluster, and merge clusters until a stopping criterion is met. divisive: start with all points in a single cluster, and split clusters until a stopping criterion is met.

Algorithm Examples



For **Simulated Annealing**, with an initial temperature $T_0 = 3.0$, cooling rate $\alpha = 0.7$ per iteration, at state s with heuristic $h(s)$, we pick a random neighbour s', compute $\Delta = h(s') - h(s)$. If $\Delta \leqslant 0$, always move to s', else move with probability $\exp(-\Delta/T)$. Update $T \leftarrow \alpha T$. Repeat until goal.

Iteration 0 - Current node A, $h(A) = 4$, $T = 3.0$, random neighbour (in this case $s' = C$) $h(s') = 4$, $\Delta = 0$, accept? Yes, as equal/improve ($e^{-0/3} = 0$). Iteration 1 - Current node C, $h(s) = 4$, $T = 2.1$, random neighbour $h(s') = h(A) = 4$, $\Delta = 0$, hence accept ($e^{-\Delta/2.1} = 0$). Continue until goal.

For **Beam Search** at $k = 2$, keep the 2 best nodes per depth according to $h(n)$. On the graph above:

- Step 0: beam $\{A(4)\}$.
- Step 1: expand $A \rightarrow \{B(2), C(4)\} \rightarrow$ sorted by h: $[B, C] \rightarrow$ keep $\{B, C\}$.
- Step 2: expand $B, C \rightarrow \{D(1), E(2), D(1)\} \rightarrow$ candidates $\{D, D, E\} \rightarrow$ sorted $[D, D, E] \rightarrow$ keep $\{D, E\}$.
- Step 3: expand $D, E \rightarrow \{F(0), F(0)\} \rightarrow$ keep $\{F\}$ (goal).

Resulting path is $A \rightarrow B \rightarrow D \rightarrow F$.

Outlook	Temp	Humidity	Windy	Play?
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcst	Hot	High	False	Yes
Rain	Mild	High	False	Yes
Rain	Cool	Normal	False	Yes

For **Naïve Bayes**, given the instance (Sunny, Cool, High, False) we first calculate the priors ($P(Yes) = \frac{3}{5}$, $P(No) = \frac{2}{5}$). Given the k-values per attribute (3 for outlook, 3 for temp, 2 humidity, 2 windy), we can use Laplace smoothing. Given the formula:

$$P(a = v \mid C) = \frac{count(a = v, C) + 1}{N_C + k}$$

where N_C = # of examples in class C, we can calculate for both priors. For $P(\cdot \mid Yes)$ where $Outlook = Sunny$, $(0 + 1)/(3 + 3) = 1/6$, where $Temp = Cool$, $(1 + 1)/(3 + 3) = 2/6$, etc. for both Yes and No. Then, the product of the priors and all Yes/No attributes of the instance is the final score (respective). If a score No is larger than score Yes, predict No, and vice-versa.

For a **Decision Tree**, we first calculate root entropy, that is $p_{Yes} = 3/5$ and $p_{No} = 2/5$, to find $H(S) = -3/5 \log_2 3/5 - 2/5 \log_2 2/5 \approx 0.971$. We than calculate the information gain for each attribute. For Outlook, it is very simple, as each attribute lines up (Sunny has both No, Rain has both Yes), hence weighted entropy is 0 and the gain is $0.971 - 0 = 0.971$. However, for Windy, we find $p_{False} = 3/4$ as we have one No and three Yes. This means $H(Windy = False) = -1/4 \log_2 1/4 - 3/4 \log_2 3/4 \approx 0.811$ and $weight = 4/5$, hence the weighted $H = (4/5) \times 0.811 = 0.649$, making the $Gain = 0.971 - 0.649 = 0.322$. Ultimately, Outlook has highest gain, so we make a decision tree on Outlook.

A **Multi-Layered Perceptron**, given two inputs x_1, x_2 plus a bias, two hidden neurons h_1, h_2 (with bias), and a single output neuron o, and where all activations are sigmoid:

$$\sigma(v) = \frac{1}{1 + e^{-v}}$$

can pass an echelon as follows. Where $x_1 = 0.05, x_2 = 0.10, x_0\ (bias) = 1$, the target $t = 0.01$, learning rate $\eta = 0.5$, and initial weights/biases:

$$w_{1h_1} = 0.15, \quad w_{2h1} = 0.20, \quad b_{h1} = 0.35$$

$$w_{1h_2} = 0.25, \quad w_{2h2} = 0.30, \quad b_{h2} = 0.35$$

$$w_{h_1,o} = 0.40, \quad w_{h2,o} = 0.45, \quad b_o = 0.60$$

We can **forward pass** by finding the hidden layer nets and outputs:

$$net_{h1} = 0.05 \cdot 0.15 + 0.10 \cdot 0.20 + 0.35 = 0.3775$$

$$h_1 = \sigma(0.3775) \approx 0.59327$$

$$net_{h2} = 0.05 \cdot 0.25 + 0.10 \cdot 0.30 + 0.35 = 0.3925$$

$$h_2 = \sigma(0.3925) \approx 0.59688$$

the output net and output:

$$net_o = 0.59327 \cdot 0.40 + 0.59688 \cdot 0.45 + 0.60 \approx 1.1059$$

$$o = \sigma(1.1059) \approx 0.75136$$

and the error:

$$E = \frac{1}{2}(t - o)^2 = \frac{1}{2}(0.01 - 0.75136)^2 \approx 0.2748$$

We then form our **backward pass** using the deltas, where the output delta is:

$$\delta_o = (o - t) \cdot o \cdot (1 - o) = (0.75136 - 0.01) \times 0.75136 \times 0.24864 \approx 0.1370$$

we then use this to calculate the hidden deltas:

$$\delta_{h_1} = \delta_o \times w_{h_1,o} \times h_1(1 - h_1) \approx 0.1370 \times 0.40 \times 0.59327 \times 0.40673 \approx 0.0130$$

$$\delta_{h_2} = \delta_o \times w_{h_2,o} \times h_2(1 - h_2) \approx 0.1370 \times 0.45 \times 0.59688 \times 0.40312 \approx 0.0140$$

We then must **update our weights** by applying $w' \leftarrow w - \eta(\delta \times input)$, which is also biased (but since $bias = 1$, nothing). For our output layer:

$$w'_{1,h_1} = 0.15 - 0.5 \times 0.0130 \times 0.05 \approx 0.1497$$
$$w'_{2,h_1} = 0.20 - 0.5 \times 0.0130 \times 0.10 \approx 0.1994$$
$$b'_{h_1} = 0.35 - 0.5 \times 0.0130 \times 1 \approx 0.3435$$
$$w'_{1,h_2} = 0.25 - 0.5 \times 0.0140 \times 0.05 \approx 0.2497$$
$$w'_{2,h_2} = 0.30 - 0.5 \times 0.0140 \times 0.10 \approx 0.2993$$
$$b'_{h_2} = 0.35 - 0.5 \times 0.0140 \times 1 \approx 0.3430$$

All these steps is equivalent to a single epoch in such a Multi-Layered Perceptron (jesus christ). When repeated, the it will progressively get closer to t.

To make a maximum-margin classifier (Hard-Margin **Scalar Vector Machine**) for the dataset $P_1 = (1, 1), P_2 = (-1, -1)$, we first find the primal problem - that is, to find $\boldsymbol{w} \in \mathbb{R}^2$ and b minimising $\frac{1}{2}\|\boldsymbol{w}\|^2$ subject to the constraints for each i: $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geqslant 1$. For our two points, this gives:

$$\begin{cases} (+1)(w_1 \cdot 1 + w_2 \cdot 1 + b) \geqslant 1, & (P_1) \\ (-1)(w_1 \cdot (-1) + w_2 \cdot (-1) + b) \geqslant 1 & (P_2) \end{cases}$$

At optimum, both constraints hold with equality (both points become support vectors):

$$1.\ w_1 + w_2 + b = 1$$

$$2.\ -(-w_1 - w_2 + b) = 1 \implies -w_1 - w_2 + b = -1$$

Adding the two equations:

$$(w_1 + w_2 + b) + (-w_1 - w_2 + b) = 1 + (-1)$$

$$\implies 2b = 0 \implies b = 0$$

Substitute $b = 0$ into $w_1 + w_2 = 1$. We still have infinitely many (w_1, w_2) on that line, so we pick the one minimising $\|\boldsymbol{w}\|^2 = w_1^2 + w_2^2$. By symmetry (and Lagrange multipliers) the minimum occurs at $w_1 = w_2 = \frac{1}{2}$. Thus $\boldsymbol{w} = (\frac{1}{2}, \frac{1}{2}), \quad b = 0$.

We can then find the decision boundary where $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \implies \frac{1}{2}x_1 + \frac{1}{2}x_2 = 0$, or simply $x_1 + x_2 = 0$. The margin is where

$$\gamma = \frac{1}{\|\boldsymbol{w}\|} = \frac{1}{\sqrt{(1/2)^2 + (1/2)^2}} = \frac{1}{\sqrt{1/2}} = \sqrt{2}$$

Approaches to Questions

Two main approaches, Constraint Satisfaction Problems and STRIPS.

CSP asks: 'Can we assign values to a set of variables so that all constraints are satisfied?'. It is formed by its' variables $X_1, X_2, \dots, X_n$, domains D_i for each X_i, and constraints C_{ij} between variables (binary) or more generally $C_{i_1}, C_{i_2}, \dots, C_{i_n}$. The solution is an assignment $X_1 = v_1, ..., X_n = v_n$ that makes every constraint true. To check, perform a backtracking search, picking an unassigned variable and assigning it from its domain, then, if no constraints are violated, recurse, else backtrack and try another value. Once you assign $X_i = v$, immediately remove any values w from each neighbours domain if (v, w) violates the constraint. One may also follow Arc Consistency (AC-3), where one initialises a queue of all arcs (X_i, X_j), then while the queue is not empty, pop (X_i, X_j). For each $v \in D_i$, if there is no $w \in D_j$ with $(v, w) \in C_{ij}$, remove v from D_i and enqueue all (X_k, X_i) for neighbours X_k. Note you should always choose the variable with the fewest legal values left, breaking these ties by picking the variable in more constraints, and when assigning, preferring the value that rules out the fewest choices for neighbours.

STRIPS planning problems follow: 'Given an initial state and a goal description, find a sequence of actions that achieves the goal'. A state s is a set of grounded predicates (e.g. {At A, Free B}). Each action a is a tuple (Pre, Add, Del), where $Pre \subseteq$ predicates that must hold in s, Add are the predicates to be added to the state, and Del are predicates to be removed from the state. If $Pre \subseteq s$, then a is applicable and $Result(s, a) = (sn Del) \cup Add$. One might choose to take a regressive (start from goal conditions and regress) or progressive (from initial state, applying actions) search to solve a problem. It is typically a smart idea to build a graph with alternating layers $\langle S_0, A_0, S_1, A_1, \dots \rangle$ where S_i is the set of predicates reachable after i steps, and A_i is the set of actions whose preconditions lie in S_i. Ensure to track mutexes (actions/literals that cannot co-occur, comp2017 moment) and ensure to stop when all goal literals appear in S_k without mutual exclusion. One can then extract a plan by choosing a non-mutex action for each goal at level k, then regressing subgoals to level $k - 1$, and so on.