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: $O(b^{[C*/\epsilon]})$ 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)

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 lowest $h(n)$, but only considers successors of the current node. solution found depends on the initial node. hill climbing finds the local minimum/maximum. complete: no (local maxima, plateaus, ridges) optimal: no (local maxima, plateaus, ridges)

Beam Search: keeps the k best nodes 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.

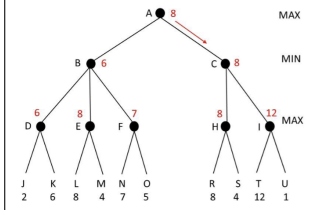
mutation is random change of bits to an individual.

complete/optimal: no

Games

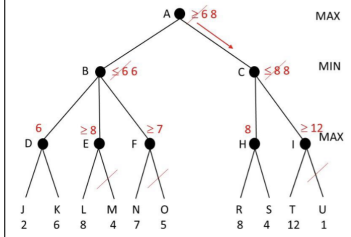
deterministic vs chance, e.g. dice vs no dice games perfect vs imperfect information: e.g. chess vs poker zero-sum vs non-zero-sum: e.g. one's gain is another's loss vs both can win process forward, not from goal, too many possible goals, don't learn anything if incomplete search from goal

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 parrent prune



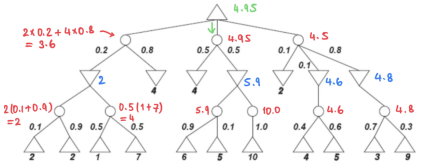
Imperfect minmax and alpha-beta pruning takes too long for large trees, so we cut off the search at a certain depth, and evaluate the nodes using a heuristic function. however with cutoff depth we may cutoff before a losing move so we do a secondary search past the cutoff point to ensure that there are no hidden pitfalls. non-quiescent moves are 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 - generalsation of euclidean and manhattan

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

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 x_i are the features of the example, 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.

$Accuracy = 1 - Errorrate$

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.

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

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

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)

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

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

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

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

(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_j is the subset of examples in S that have value j for feature A, and $values(A)$ is the set of all possible values for feature A.

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

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 autoencoders 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 misclassifcation 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

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

where b is the bias term, which can be calculated as:

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 decisio tree is made. no pruning.

Bayesian Networks

these are used to represent the probabilistic relationships between variables.

computing join probability BN

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

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 Burglar (B):

	b	~b
s	0.25	0.75
~s	0.33	0.67

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

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

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

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

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

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

$= 0.31 * \sum_b P(B_b|s)[0.75 * P(a|B_b,c) + 0.25 * 0.25 * P(a|B_b,\sim c)] =$ <- 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. reapeat until the centroids do not change or a maximum number of iterations is reached.

Neartest 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.