CBCS SCHEME

USN 1 C R 2 2 A I 0 1 4

BAI702

Seventh Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026
Machine Learning - II

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain the fundamental design issues and approaches considered while developing a machine learning program for playing checkers game.	10	L2	CO1
	b.	Describe the key perspectives and issues in machine learning.	04	L2	CO1
	c.	State and prove the Version space representation theorem. Clearly show that : i) Every hypothesis 'h' satisfying $(\exists s \in S)\ (\exists g \in G)\ (g \geq_g h \geq s)$ is a member of $VS_{H,D}$ ii) Every hypothesis in $VS_{H,D}$ satisfies this condition.	06	L3	CO1
		OR			
Q.2	a.	Describe well – posed learning problems in machine learning with suitable examples.	05	L2	CO1
	b.	Explain Enjoy Sport concept learning task.	05	L2	CO1
	c.	Outline the steps involved in the candidate – Elimination algorithm for computing version space and explain with an example.	10	L3	CO1
		Module – 2			
Q.3	a.	Explain general – to – specific beam search and LEARN –ONE –RULE algorithm.	10	L2	CO2
	b.	Using a small relational dataset, explain the FOIL algorithm and apply the same to generate first – order rule.	10	L3	CO2
		OR			
Q.4	a.	Outline the explanation –based learning algorithm PROLOG – EBG with help of SafeToStack example.	10	L3	CO2
	b.	Describe how explanation based learning can be used to derive search control knowledge.	06	L2	CO2
	c.	List the three different methods for using prior knowledge to alter the search performed by purely inductive methods.	04	L2	CO2
		Module – 3			
Q.5	a.	Explain the AdaBoost algorithm and how it sequentially improves weak classifiers.	08	L2	CO2
	b.	Outline the steps of basic random forest training algorithm.	04	L2	CO2
	c.	Explain the K – means algorithm with an example.	08	L2	CO2
		OR			
Q.6	a.	Explain ensemble learning and why combining multiple models often improves performance compared to a single model.	06	L2	CO3
	b.	Explain the K – Means neural network algorithm.	08	L2	CO3
	c.	Outline the steps of the mixture of experts algorithm.	06	L2	CO3

Module – 4

Q.7	a.	Describe the vector quantization with diagram that shows an interpretation of prototype vectors in two dimensions.	04	L2	CO3
	b.	Illustrate the self organizing feature map algorithm.	08	L3	CO3
	c.	Explain the following with respect to self organizing feature map. i) Self – Organization ii) Network dimensionality and boundary conditions.	08	L2	CO3
		OR			
Q.8	a.	Discuss the following : i) The Box – Muller scheme ii) Monte Carlo Principle.	10	L2	CO4
	b.	Describe Markov Chain Monte Carlo (MCMC) and explain Metropolis – Hastings algorithm used for MCMC.	10	L3	CO4
		Module – 5			
Q.9	a.	Consider the Bayesion network as shown in fig. Q. 9 (a) and identify the following. i) The probability of being scared ii) The probability the student revised given that they are scared iii) The probability they attended given that they are scared.	10	L3	CO4
	b.	Explain the Hidden Markov Models (HMMs) Forward algorithm and HMM Viterbi algorithm.	10	L2	CO4
		OR			
Q.10	a.	Illustrate the steps of Kalman filter algorithm and particle filter algorithm.	10	L3	CO4
	b.	Explain the Baum – Welch algorithm.	10	L2	CO4

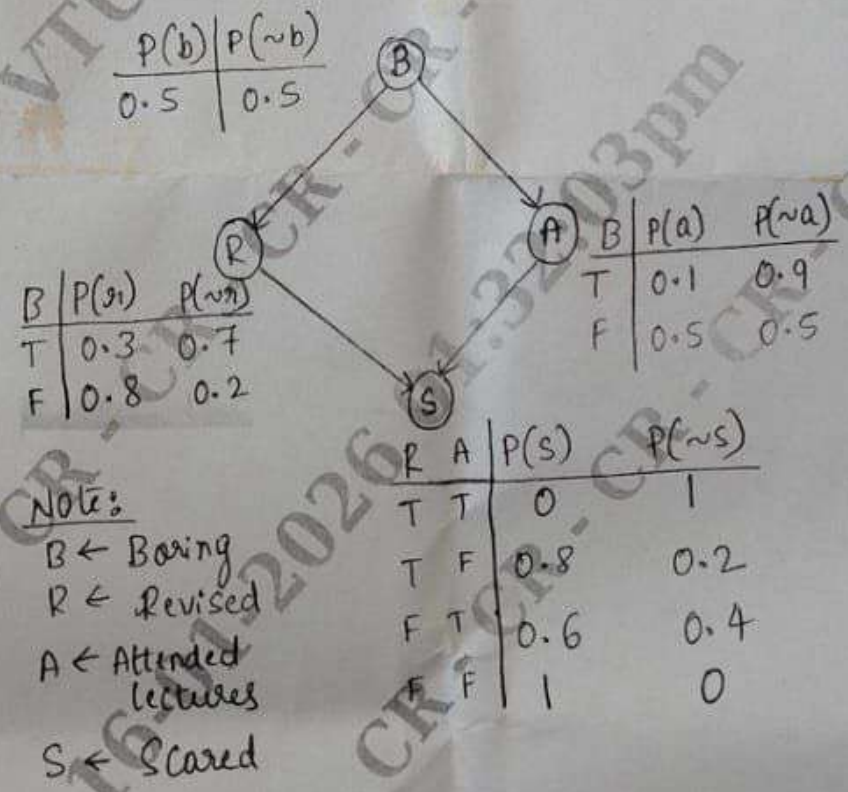


Fig. Q. 9(a)

Q1(a) Fundamental design issues & approaches for ML program playing Checkers (10M)

Fundamental Design Issues:

1. **Choice of Training Experience**
    - Self-play games
    - Games played by experts
    - Recorded past games
2. **Target Function**
    - Learn a function V(b) that evaluates a board position b
    - Output: probability of winning or utility value
3. **Representation of Target Function**
    - Linear combination of features:
      V(b)=w0+w1f1(b)+w2f2(b)+⋯+wnfn(b)V(b) = w_0 + w_1f_1(b) + w_2f_2(b) + \dots + w_nf_n(b)V(b)=w0+w1f1(b)+w2f2(b)+⋯+wnfn(b)
    - Features: number of pieces, kings, threatened pieces, etc.
4. **Learning Algorithm**
    - Gradient Descent
    - Temporal Difference Learning (TD Learning)
5. **Performance Measure**
    - Winning percentage against opponents

Approaches Considered:

- Supervised learning

- Reinforcement learning
- Temporal difference methods

Q1(b) Key perspectives and issues in ML (4M)

Perspectives:

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner. For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights wo through w6. The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works well when the hypothesis representation.considered by the learner defines a continuously parameterized space of potential. These different hypothesis representations are appropriate for learning different kinds of target functions. For each of these hypothesis representations, the corresponding learning algorithm takes advantage of a different underlying structure to organize the search through the hypothesis space. We will also find this viewpoint useful in formally analyzing the relationship between the size of the hypothesis space to be searched, the number of training examples available, and the confidence we can have that a hypothesis consistent with the training data will correctly generalize to unseen examples.

Issues:

- What algorithms exist for learning general target functions from specific training examples. In what settings will particular algorithms converge to the desired function, given sufficient training data.Which algorithms perform best for which types of problems and representations.
- How much training data is sufficient. What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space.
- When and how can prior knowledge held by the learner guide the process of generalizing from examples.Can prior knowledge be helpful even when it is only approximately correct.
- What is the best strategy for choosing a useful next training experience, and

how does the choice of this strategy alter the complexity of the learning problem.

- What is the best way to reduce the learning task to one or more function approximation problems. Put another way, what specific functions should the system attempt to learn. Can this process itself be automated.
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function.

Q1(c) Version Space Representation Theorem (6M)

Statement:

The version space VS_{H,D} is exactly the set of hypotheses consistent with all training examples.

Proof:

i) **Every hypothesis h satisfying**

(∃s∈S)(∃g∈G)(g≥h≥s)(\exists s \in S)(\exists g \in G)(g \ge h \ge s)(∃s∈S)(∃g∈G)(g≥h≥s)

is in VS_{H,D}.

- Since s and g are consistent with data,
- Any hypothesis between them is also consistent
- Hence h ∈ VS_{H,D}

ii) **Every hypothesis in VS_{H,D} satisfies this condition**

- For any consistent hypothesis h,
- There exists a most specific hypothesis s
- And a most general hypothesis g
- Such that g ≥ h ≥ s

Q2(a) Well-posed learning problem with example (5M)

Definition: A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself. In general, to have a well-defined learning problem, we must identity these three features: the class of tasks, the measure of performance to be improved, and

the source of experience.

A checkers learning problem:

Task T: playing checkers
Performance measure P: percent of games won against opponents
Training experience E: playing practice games against itself

A handwriting recognition learning problem:

Task T: recognizing and classifying handwritten words within images
Performance measure P: percent of words correctly classified
Training experience E: a database of handwritten words with given classifications.

A robot driving learning problem:
Task T: driving on public four-lane highways using vision sensors
Performance measure P: average distance traveled before an error (as judged by human overseer)
Training experience E: a sequence of images and steering commands recorded while observing a human driver

Q2(b) EnjoySport concept learning task (5M)

Given:
Instances X: Possible days, each described by the attributes
Sky (with possible values Sunny, Cloudy, and Rainy),

AirTemp (with values Warm and Cold),
Humidity (with values Normal and High),
Wind (with values Strong and Weak),
Water (with values Warm and Cool), and
Forecast (with values Same and Change).

Hypotheses H: Each hypothesis is described by a conjunction of constraints on the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. The constraints may be "?" (any value is acceptable), "
0" (no value is acceptable), or a specific value.

Target concept c: EnjoySport : X -> {0,l}.

Training examples D: Positive and negative examples of the target function .

Determine:
A hypothesis h in H such that h(x) = c(x) for all x in X.

Q2(c) Candidate Elimination Algorithm (10M)

Steps:

1.Initialize G to the set of maximally general hypotheses in H
2.Initialize S to the set of maximally specific hypotheses in H
3.For each training example d, do
If d is a positive example
Remove from G any hypothesis inconsistent with d ,
For each hypothesis s in S that is not consistent with d ,-
Remove s from S
Add to S all minimal generalizations h of s such that
h is consistent with d, and some member of G is more general than h
Remove from S any hypothesis that is more general than another hypothesis in S

4. If d is a negative example
Remove from S any hypothesis inconsistent with d
For each hypothesis g in G that is not consistent with d
Remove g from G
Add to G all minimal specializations h of g such that
h is consistent with d, and some member of S is more specific than h

Remove from G any hypothesis that is less general than another hypothesis in G

MODULE-2

3a)General-to-Specific Beam Search & Learn-One-Rule (10M)

LEARN-ONE-RULE is a rule-based learning algorithm that searches the hypothesis space in a manner similar to the ID3 decision tree algorithm. It begins with the most general rule having an empty precondition that matches all training examples.

The algorithm performs a general-to-specific greedy search by successively adding attribute–value tests that most improve the rule's performance on the training data. At each step, only the best-performing attribute test is selected, unlike ID3 which expands all branches.

LEARN-ONE-RULE(*Target_attribute, Attributes, Examples, k*)

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

- Initialize *Best_hypothesis* to the most general hypothesis ∅
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
  1. *Generate the next more specific candidate_hypotheses*
     - *All_constraints* ← the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
     - *New_candidate_hypotheses* ←
       for each h in *Candidate_hypotheses*,
       for each c in *All_constraints*,
       - create a specialization of h by adding the constraint c
     - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
  2. *Update Best_hypothesis*
     - For all h in *New_candidate_hypotheses* do
       - If (PERFORMANCE(h, *Examples*, *Target_attribute*)
         > PERFORMANCE(*Best_hypothesis*, *Examples*, *Target_attribute*))
         Then *Best_hypothesis* ← h
  3. *Update Candidate_hypotheses*
     - *Candidate_hypotheses* ← the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
  "IF *Best_hypothesis* THEN *prediction*"
  where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h, *Examples*, *Target_attribute*)

- *h_examples* ← the subset of *Examples* that match h
- return −*Entropy*(*h_examples*), where entropy is with respect to *Target_attribute*

3b) FOIL algorithm (10M)

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do
  - *Learn a NewRule*
    - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
    - *NewRuleNeg* ← *Neg*
    - while *NewRuleNeg*, do
      - *Add a new literal to specialize NewRule*
        - *Candidate_literals* ← generate candidate new literals for *NewRule*, based on *Predicates*
        - *Best_literal* ← $\underset{L \in Candidate\_literals}{\text{argmax}}$ $Foil\_Gain(L, NewRule)$
        - add *Best_literal* to preconditions of *NewRule*
        - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
    - *Learned_rules* ← *Learned_rules* + *NewRule*
    - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

4a)Explanation based learning PROLOG-EBG Algorithm

PROLOG-EBG (Kedar-Cabelli and McCarty, 1987) is an explanation-based learning (EBL) algorithm that represents a class of similar EBL methods. It operates as a sequential covering algorithm, learning one Horn clause rule at a time.

The algorithm learns a single rule that covers some positive training examples, removes the covered examples, and then repeats this process until all positive examples are covered. Given a complete and correct domain theory, PROLOG-EBG is guaranteed to generate a hypothesis that is logically correct and consistent with the training data.

The hypothesis produced by PROLOG-EBG represents a set of logically sufficient conditions for the target concept, derived using the domain theory. PROLOG-EBG is a refinement of the original EBG algorithm proposed by Mitchell et al. (1986) and is closely related to the EGGS algorithm by DeJong and Mooney (1986).

PROLOG-EBG(*TargetConcept*, *TrainingExamples*, *DomainTheory*)

- *LearnedRules* ← {}
- *Pos* ← the positive examples from *TrainingExamples*
- for each *PositiveExample* in *Pos* that is not covered by *LearnedRules*, do
  1. *Explain:*
     - *Explanation* ← an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*
  2. *Analyze:*
     - *SufficientConditions* ← the most general set of features of *PositiveExample* sufficient to satisfy the *TargetConcept* according to the *Explanation*.
  3. *Refine:*
     - *LearnedRules* ← *LearnedRules* + *NewHornClause*, where *NewHornClause* is of the form

$$TargetConcept \leftarrow SufficientConditions$$

- Return *LearnedRules*

4b) "Describe how explanation based learning can be used to derive search control knowledge."

Explanation-based learning (EBL), exemplified by the PROLOG-EBG algorithm, is a method where a system learns general rules from a single example by using a correct and complete domain theory. Its practical application is most feasible in domains where such a theory exists, particularly for learning to control search in complex problems, such as game playing, scheduling, or optimization. In these problems, the set of possible states (S), legal operators (O), and goal conditions (G) form a complete description, allowing the system to learn which operator or action moves a state closer to the goal. The learning task can be formulated by defining target concepts for each operator, such as "the set of states for which this operator advances toward a goal."

A concrete example is the PRODIGY system, a domain-independent planning system that uses a means-ends planning strategy. PRODIGY decomposes problems into subgoals and repeatedly faces choices like "Which subgoal should be solved next?" or "Which operator should be applied?" By encountering conflicts during search, PRODIGY generates explanations for why certain sequences lead to inefficiency and extracts rules to prevent such conflicts. For instance, in a block-stacking problem, it may learn the rule:

IF one subgoal is On(x, y) and another is On(y, z), THEN solve On(y, z) before On(x, y).

This rule prevents the system from undoing previous subgoal solutions, demonstrating how EBL uses domain-independent knowledge of conflicts together with domain-specific operator knowledge to improve efficiency. Similarly, the SOAR architecture generalizes this approach, using a variant called chunking to learn from search impasses and generate rules for future decisions.

Despite these successes, there are practical challenges. First, the number of learned rules can grow very large, making rule matching computationally expensive. Methods such as utility-based rule selection—learning only rules with benefits exceeding their costs and pruning low-utility rules—help manage this problem, as shown in PRODIGY and SOAR. Second, constructing full explanations for target concepts can be intractable in complex domains like chess, because proving an operator leads to optimal outcomes requires evaluating all alternatives. To address this, incremental or lazy explanation methods produce partial explanations and refine rules based on performance feedback, allowing the system to learn efficiently without exhaustive reasoning.

EBL has been successfully applied to a variety of domains, from simple block-stacking and planning problems to integration with reinforcement learning, demonstrating its potential to acquire search control knowledge efficiently while balancing rule complexity and computational cost.

Q5(a) AdaBoost algorithm (8M)

Idea: Combine multiple weak classifiers into a strong classifier.

Algorithm Steps:

AdaBoost Algorithm

- Initialise all weights to $1/N$, where N is the number of datapoints
- While $0 < \epsilon_t < \frac{1}{2}$ (and $t < T$, some maximum number of iterations):
  - train classifier on $\{S, w^{(t)}\}$, getting hypotheses $h_t(x_n)$ for datapoints x_n
  - compute training error $\epsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n))$
  - set $\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  - update weights using:

$$w_n^{(t+1)} = w_n^{(t)} \exp(\alpha_t I(y_n \neq h_t(x_n))/Z_t,$$

    where Z_t is a normalisation constant
- Output $f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

The AdaBoost algorithm is conceptually very simple. At each iteration a new classifier is trained on the training set, with the weights that are applied to the training set for each datapoint being modified at each iteration according to how successfully that datapoint has been classified in the past. The weights are initially all set to the same value, 1/N, where N is the number of datapoints in the training set. Then, at each iteration, the error () is computed as the sum of the weights of the misclassified points, and the weights for incorrect examples are updated by being multiplied by α = (1 −)/. Weights for correct examples are left alone, and then the whole set is normalised so that it sums to 1 (which is effectively a reduction in the importance of the correctly classified datapoints). Training terminates after a set number of iterations, or when either all of the datapoints are classified correctly, or one point contains more than half of the available weight.

Q5(b) Random Forest training steps (4M)

• For each of N trees:

– create a new bootstrap sample of the training set

– use this bootstrap sample to train a decision tree

– at each node of the decision tree, randomly select m features, and compute the information gain (or Gini impurity) only on that set of features, selecting the optimal one

– repeat until the tree is complete

Q5(c) K-Means algorithm (8M)

The k-Means Algorithm

- **Initialisation**
  - choose a value for k
  - choose k random positions in the input space
  - assign the cluster centres $\boldsymbol{\mu}_j$ to those positions
- **Learning**
  - repeat
    - for each datapoint $\mathbf{x}_i$:
      - compute the distance to each cluster centre
      - assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j). \tag{14.1}$$

    - for each cluster centre:
      - move the position of the centre to the mean of the points in that cluster (N_j is the number of points in cluster j):

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N_j} \mathbf{x}_i$$

  - until the cluster centres stop moving
- **Usage**
  - for each test point:
    - compute the distance to each cluster centre
    - assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j).$$

Q6(a) Ensemble learning explanation (6M)

Definition:

The old saying has it that two heads are better than one. Which naturally leads to the idea that even more heads are better than that, and ends up with decision by committee, which is famously useless for human activities (as in the old joke that a camel is a horse designed by a committee). For machine learning methods the results are rather more impressive, as we'll see in this chapter.

The basic idea is that by having lots of learners that each get slightly different results on a dataset—some learning certain things well and some learning others—and putting them together, the results that are generated will be significantly better than any one of them on its own (provided that you put them together well... otherwise the results could be significantly worse). One analogy that might prove useful is to think about how your doctor goes about performing a diagnosis of some complaint that you visit her with. If she cannot find the problem directly, then she will ask for a variety of tests to be performed, e.g., scans, blood tests, consultations with experts. She will then aggregate all of these opinions in order to perform a diagnosis. Each of the individual tests will suggest a diagnosis, but only by putting them together can an informed decision be reached.

Figure 13.1 shows the basic idea of ensemble learning, as these methods are collectively called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

Combining multiple models to improve accuracy.

Why it works:

- Reduces variance
- Reduces bias
- Improves robustness

Examples: Bagging, Boosting, Random Forest

Q6(b) K-Means Neural Network (8M)

The On-Line k-Means Algorithm

- **Initialisation**
  - choose a value for k, which corresponds to the number of output nodes
  - initialise the weights to have small random values
- **Learning**
  - normalise the data so that all the points lie on the unit sphere
  - repeat:
    - for each datapoint:
      - compute the activations of all the nodes
      - pick the winner as the node with the highest activation
      - update the weights using Equation (14.7)
    - until number of iterations is above a threshold
- **Usage**
  - for each test point:
    - compute the activations of all the nodes
    - pick the winner as the node with the highest activation

- Competitive learning
- Winner-takes-all
- Update only winning neuron
- Equivalent to K-Means clustering

Q6(c) Mixture of Experts algorithm (6M)

The Mixture of Experts Algorithm

- For each expert:
  - calculate the probability of the input belonging to each possible class by computing (where the $\mathbf{w}_i$ are the weights for that classifier):

$$o_i(\mathbf{x}, \mathbf{w}_i) = \frac{1}{1 + \exp(-\mathbf{w}_i \cdot \mathbf{x})}. \tag{13.6}$$

- For each gating network up the tree:
  - compute:

$$g_i(\mathbf{x}, \mathbf{v}_i) = \frac{\exp(\mathbf{v}_i \mathbf{x})}{\sum_l \exp(\mathbf{v}_l \mathbf{x})}. \tag{13.7}$$

- Pass as input to the next level gates (where the sum is over the relevant inputs to that gate):

$$\sum_k o_j g_j. \tag{13.8}$$

Steps:

1. Divide problem into sub-tasks
2. Train expert networks
3. Train gating network
4. Combine expert outputs
5. Weighted final prediction

Q7 a. Describe the vector quantization with diagram that shows an interpretation of prototype vectors in two dimensions.

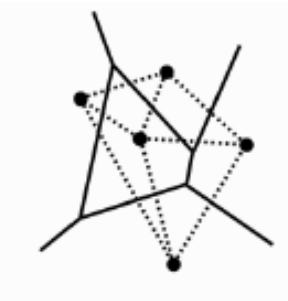


FIGURE 14.5 The Voronoi tesselation of space that performs vector quantisation. Any datapoint is represented by the dot within its cell, which is the prototype vector.

The dots at the centre of each cell are the prototype vectors, and any datapoint that lies within a cell is represented by the dot. The name for each cell is the Voronoi set of a particular prototype.

Together, they produce the Voronoi tesselation of the space. If you connect together every pair of points that share an edge, as is shown by the dotted lines, then you get the Delaunay triangulation, which is the optimal way to organise the space to perform function approximation.

We need to choose prototype vectors that are as close as possible to all of the possible inputs that we might see. This application is called learning vector quantisation because we are learning an efficient vector quantisation. The k-means algorithm can be used to solve the problem if we know how large we want our codebook to be.

b. Illustrate the self organizing feature map algorithm.

The Self-Organising Feature Map (SOM) Algorithm

Initialisation

Choose a size (number of neurons) and number of dimensions **d** for the map. Either choose random values for the weight vectors so that they are all different, or set the weight values to increase in the direction of the first **d** principal components of the dataset.

Learning

Repeat the following steps:

For each datapoint, select the **best-matching neuron** n_b using the minimum Euclidean distance between the weights and the input:

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \quad (14.8)$$

Update the weight vector of the best-matching node using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)(\mathbf{x} - \mathbf{w}_j^T) \quad (14.9)$$

where $\eta(t)$ is the learning rate.

Update the weight vectors of all other neurons using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)\, h(n_b, t)\, (\mathbf{x} - \mathbf{w}_j^T) \quad (14.10)$$

where $\eta_n(t)$ is the learning rate for neighbourhood nodes and $h(n_b, t)$ is the neighbourhood function, which determines whether each neuron belongs to the neighbourhood of the winning neuron (with $h = 1$ for neighbours and $h = 0$ for non-neighbours).

Reduce the learning rates and adjust the neighbourhood function, typically using:

$$\eta(t+1) = \alpha\eta(t)^{k/k_{\max}}, \quad 0 \leq \alpha \leq 1$$

where k is the number of iterations completed and $k_{\max}$ is the maximum number of iterations. The same equation is applied to both learning rates $\eta(t), \eta_n(t)$ and the neighbourhood function $h(n_b, t)$.

The learning process continues until the map stops changing or a maximum number of iterations is reached.

Usage

For each test datapoint, select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input:

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \quad (14.11)$$

c.Explain the following with respect to self organizing feature map.

i) Self-organization

The term self-organisation in Self-Organising Maps (SOM) refers to the emergence of a global ordering of neurons through local interactions, without the use of any external supervisor. In an SOM, each neuron interacts only with its neighbouring neurons, and neurons that are far apart do not directly influence one another. Despite this, the network as a whole develops an ordered representation of the input space.

This phenomenon is remarkable because a global structure is formed using only local learning rules. During training, neurons that are close to each other in the network respond to similar input patterns, resulting in a topologically ordered mapping of the input data. This ability of the system to organise itself without explicit control is known as self-organisation.

Self-organisation is a fundamental concept in the broader field of complexity science, where large-scale patterns arise from simple local interactions. A common real-world example of self-organisation is the formation of bird flocks. Individual birds do not have complete information about the position of all other birds. Instead, each bird follows simple local rules, such as maintaining a fixed relative position and matching speed with nearby birds. Simulations show that these simple local interactions are sufficient to produce well-structured and coordinated flock formations.

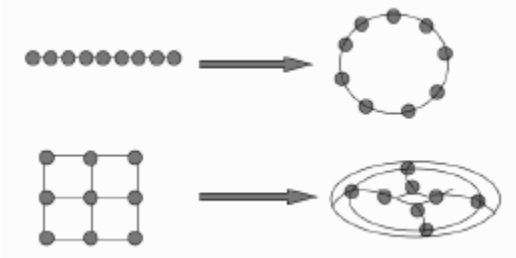
Similarly, in SOMs, the global ordering of neurons emerges naturally from local neighbourhood interactions, demonstrating how complex and organised behaviour can arise without centralized control.

ii) Network dimensionality and boundary conditions.

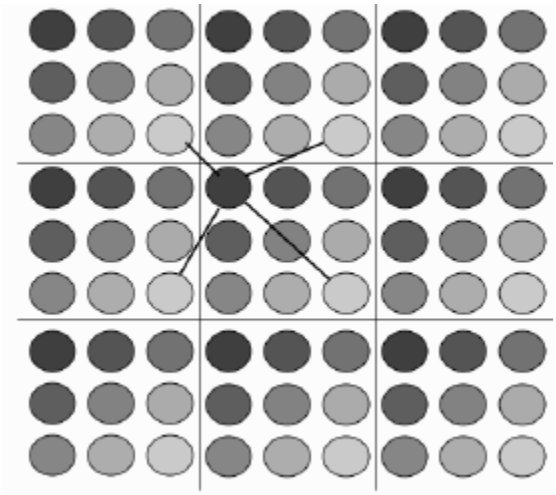
The Self-Organising Map (SOM) is usually implemented as a two-dimensional rectangular grid of neurons, but the algorithm itself does not restrict the network to two dimensions. Depending on the problem, one-dimensional, two-dimensional, or even higher-dimensional maps may be used. The appropriate dimensionality depends on the **intrinsic dimensionality** of the data, which is the minimum number of dimensions required to represent it, rather than the dimensionality of the space in which the data is embedded. Noise in real-world data often increases apparent dimensionality, and identifying intrinsic dimensionality helps reduce this effect.

Boundary conditions are also important in SOM design. In some applications, boundaries are clearly defined, such as ordering sounds from low to high pitch. In many cases, however, boundaries are not natural and can introduce edge effects. To remove these effects, the boundaries of the map can be connected. In a one-dimensional SOM this forms a circle, and in a two-dimensional SOM it forms a torus, ensuring that there are no edge neurons. Although toroidal maps often perform better than rectangular ones, distance calculations become more complex due to wrap-around effects.

The size of the SOM is fixed before learning begins and determines the granularity of representation. A small network produces coarse generalisations, while a larger network allows finer-grained learning at higher computational cost.



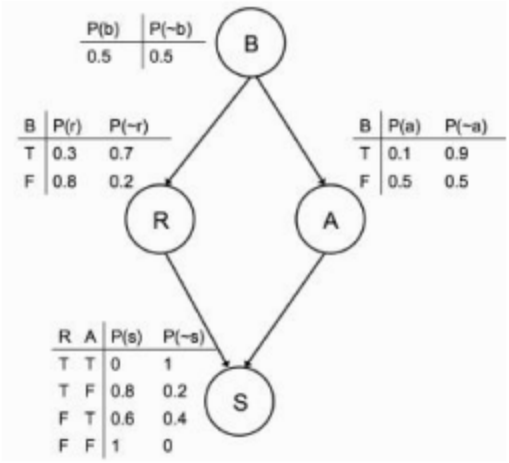
Using circular boundary conditions in 1D turns a line into a circle, while in 2D it turns a rectangle into a torus.



One way to compute distances between points without any boundary on the map is to imagine copies of the entire map being placed around the original, and picking the shortest of the istances between a node and any of the copies of the other node.

Q9(a) Bayesian Network

Given Bayesian Network



$$\begin{aligned} P(s) &= \sum_{b,r,a} P(b,r,a,s) \\ &= \sum_{b,r,a} P(b) \times P(r|b) \times P(a|b) \times P(s|r,a) \\ &= \sum_{b} P(b) \times \sum_{r,a} P(r|b) \times P(a|b) \times P(s|r,a). \end{aligned}$$

$$\begin{aligned} P(s) &= 0.3 \times 0.1 \times 0 + 0.3 \times 0.9 \times 0.8 + 0.7 \times 0.1 \times 0.6 + 0.7 \times 0.9 \times 1 \\ &= 0.328. \end{aligned}$$

$$\begin{aligned} P(r|s) &= \frac{P(s|r)P(r)}{P(s)} \\ &= \frac{\sum_{b,a} P(b,a,r,s)}{P(s)} \\ &= \frac{0.5 \cdot (0.3 \cdot 0.1 \cdot 0 + 0.3 \cdot 0.9 \cdot 0.8) + 0.5 \cdot (0.8 \cdot 0.5 \cdot 0 + 0.8 \cdot 0.5 \cdot 0.8)}{P(s)} \\ &= \frac{0.268}{0.684} = 0.3918. \\ P(a|s) &= \frac{P(s|a)P(a)}{P(s)} \\ &= \frac{0.144}{0.684} = 0.2105. \end{aligned}$$

b) Explain the Hidden Markov Models (HMMs) Forward algorithm and HMM Viterbi algorithm.

In an HMM, we have:

1. **Hidden states** S: e.g., `{partying, watching TV, studying, sleeping}`
   Let's denote them generically as $S = \{s_1, s_2, s_3, s_4\}$.
2. **Observations** O: e.g., `{tired, fine, hungover, scared}`
   Denoted $O = \{o_1, o_2, ..., o_T\}$.
3. **Initial probabilities** $\pi(s)$: probability of starting in each state.
   You mentioned assigning 0.25 to each state because we don't know the starting state.
4. **Transition probabilities** $A = P(s_t|s_{t-1})$: probability of moving from one state to another.
5. **Emission probabilities** $B = P(o_t|s_t)$: probability of observing something given the current state.

- $\alpha_{i,t}$, which is the probability of getting the observation sequence up to time t and being in state i at time t (size $N \times t$),
- $\beta_{i,t}$, which is the probability of the sequence from $t+1$ to the end given that the state is i at time t,
- $\delta_{i,t}$, which is the highest probability of any path that reaches state i at time t,
- $\xi_{i,j,t}$, which is the probability of being in state i at time t, state j at time $t+1$, and so is an $N \times N \times T$ matrix.

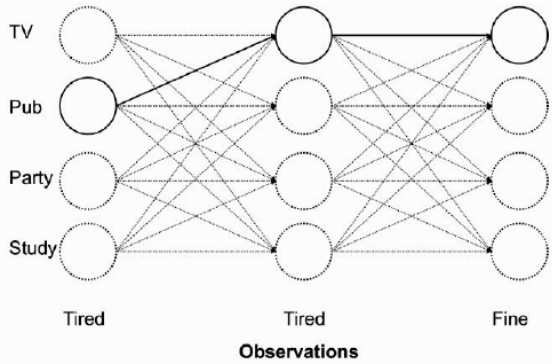
Since $\alpha_{i,t}$ is the probability of getting the observation sequence up to time t and being in state i at time t conditioned on the model and the observations, the probability of the whole observation sequence given the model is just $\sum_{i=1}^{N} \alpha_{i,T}$.

The HMM Forward Algorithm

- Initialise with $\alpha_{i,0} = \pi_i b_i(o_0)$
- For each observation in order o_t, $t = 1, \ldots, T$
  - for each of the N_s possible states s:
    * $\alpha_{s,t+1} = b_s(o_{t+1}) \left(\sum_{i=1}^{N} (\alpha_{i,t} a_{i,s}) \right)$

The HMM Viterbi Algorithm

- Start by initialising $\delta_{i,0}$ by $\pi_i bi(o_0)$ for each state i, $\phi_0 = 0$
  - run forward in time t:
    - for each possible state s:
      - $\delta_{s,t} = \max_i(\delta_{i,t-1}a_{i,s})b_s(o_t)$
      - $\phi_{s,t} = \arg\max_i(\delta_{i,t-1}a_{i,s})$
  - set q_T^*, the most likely end hidden state to be $q_T^* = \arg\max_i \delta_{i,T}$
  - run backwards in time computing:
    - $q_{t-1}^* = \phi_{q_t^*,t}$



The Viterbi trellis for the first three observations of the example HMM.

10.a)Illustrate the steps of Kalman filter algorithm and particle filter algorithm.

Once we get the actual observation z_{t+1}, we compute the **innovation (prediction error)**:

$$\text{innovation} = z_{t+1} - \hat{y}_{t+1} = z_{t+1} - H\hat{x}_{t+1}$$

- The Kalman filter then **weights this error** by how much we trust the prediction vs. the measurement. This weight is the **Kalman gain**:

$$K_{t+1} = \hat{\Sigma}_{t+1}H^T(H\hat{\Sigma}_{t+1}H^T + R)^{-1}$$

- Intuition:
  - If measurement noise R is small (we trust measurements more), K is higher → more correction from measurement.
  - If prediction uncertainty $\hat{\Sigma}$ is small (we trust our model more), K is lower → less correction.
- **State update:**

$$x_{t+1} = \hat{x}_{t+1} + K_{t+1}(z_{t+1} - H\hat{x}_{t+1})$$

- **Covariance update:**
  After the measurement, the uncertainty decreases:

$$\Sigma_{t+1} = (I - K_{t+1}H)\hat{\Sigma}_{t+1}$$

- Given an initial estimate $\mathbf{x}(0)$
- For each timestep:
  - **predict the next step**
    - predict state as $\hat{\mathbf{x}}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t$
    - predict covariance as $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$
  - **update the estimate**
    - compute the error in the estimate, $\boldsymbol{\epsilon} = \mathbf{y}_{t+1} - \mathbf{H}\mathbf{A}\mathbf{x}_{t+1}$
    - compute the Kalman gain using Equation (16.27)
    - update the state using Equation (16.28)
    - update the covariance using Equation (16.29)

Particle Filter:

- Sample $\mathbf{x}_0^{(i)}$ from $p(\mathbf{x}_0$ for $i = 1 \ldots N$
- For each timestep:
  - **importance sample**
  - for each datapoint:
    - sample $\hat{\mathbf{x}}_t^{(i)}$ from $q(\mathbf{x}_t^{(i)} | \mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})$
    - add $\hat{\mathbf{x}}_t^{(i)}$ onto the list of samples to get $\mathbf{x}_{0:t}^{(i)}$ from $\mathbf{x}_{0:t-1}^{(i)}$
    - compute the importance weights:

$$w_t^{(i)} = w_{t-1}^{(i)} \frac{p(\mathbf{y}_t | \hat{\mathbf{x}}_t^{(i)}) p(\mathbf{x}_t^{(i)} | \hat{\mathbf{x}}_{t-1}^{(i)})}{q(\mathbf{x}_t^{(i)} | \mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})} \tag{16.38}$$

  - normalise the importance weights by dividing by their sum
  - **resample the particles**
    - retain particles according to their importance weights, so that there might be several copies of some particles, and none of others to get the same number of particles approximately sampled from $p(\mathbf{x}_{0:t}^{(i)} | \mathbf{y}_{1:t})$

---

b.Explain the Baum–Welch algorithm

The HMM Baum–Welch (Forward–Backward) Algorithm

- Initialise π to be equal probabilities for all states, and a, b randomly unless you have prior knowledge
- While updates have not converged:
  - **E-step:**
  - use forward and backward algorithms to get α and β
  - for each observation in the sequence $o_t, t = 1 \ldots T$
    - for each state i:
      - for each state j:
      - compute ξ using Equation (16.20)
  - **M-step:**
  - for each state i:
    - compute $\hat{\pi}_i$ using Equation (16.17)
    - for each state j:
      - compute $\hat{a}_{i,j}$ using Equation (16.18)
  - for each different possible observation o:
    - compute $\hat{b}_i(o)$ using Equation (16.19)