UNIT – I

Machine Learning – SCSA1601

UNIT 1 INTRODUCTION TO MACHINE LEARNING

Machine learning - examples of machine learning applications - Learning associations - Classification - Regression - Unsupervised learning - Supervised Learning - Learning class from examples - PAC learning - Noise, model selection and generalization - Dimension of supervised machine learning algorithm.

I. Introduction

Well posed learning problem: "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."(Tom Michel)

"Field of study that gives computers the ability to learn without being explicitly programmed". Learning = Improving with experience at some task

- Improve over task T,
- with respect to performance measure P,
- based on experience

E.E.g., Learn to play

checkers

- T : Play checkers
- P : % of games won in world tournament
- E: opportunity to play against self

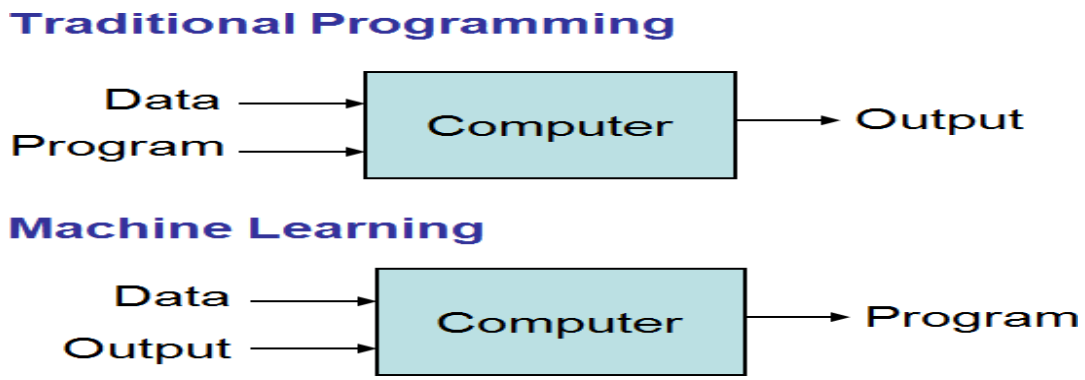


Fig. 1.1 Traditional Programming Vs Machine Learning

Examples of tasks that are best solved by using a learning algorithm:

- Recognizing patterns:
  - Facial identities or facial expressions
  - Handwritten or spoken words
  - Medical images
- Generating patterns:
  - Generating images or motion sequences (demo)
- Recognizing anomalies:
  - Unusual sequences of credit card transactions
  - Unusual patterns of sensor readings in a nuclear power plant or unusual sound inyour car engine.
- Prediction:
  - Future stock prices or currency exchange rates
- The web contains a lot of data. Tasks with very big datasets often use machine learning
  - especially if the data is noisy or non-stationary.
- Spam filtering, fraud detection:
  - The enemy adapts so we must adapt too.
- Recommendation systems:
  - Lots of noisy data. Million dollar prize!
- Information retrieval:
  - Find documents or images with similar content.
- Data Visualization:
  - Display a huge database in a revealing way

Types of learning algorithms:

- ***Supervised learning***
  - Training data includes desired outputs. Examples include,
    - Prediction
    - Classification (discrete labels), Regression (real values)

- ***Unsupervised learning***
  - Training data does not include desired outputs, Examples include,
    - Clustering
    - Probability distribution estimation
    - Finding association (in features)
    - Dimension reduction
- ***Semi-supervised learning***
  - Training data includes a few desired outputs
- ***Reinforcement learning***
  - Rewards from sequence of actions
    - Policies: what actions should an agent take in a particular situation
    - Utility estimation: how good is a state (used by policy)
  - No supervised output but delayed reward
  - Credit assignment problem (what was responsible for the outcome)
  - Applications:
    - Game playing
    - Robot in a maze
    - Multiple agents, partial observability, ...

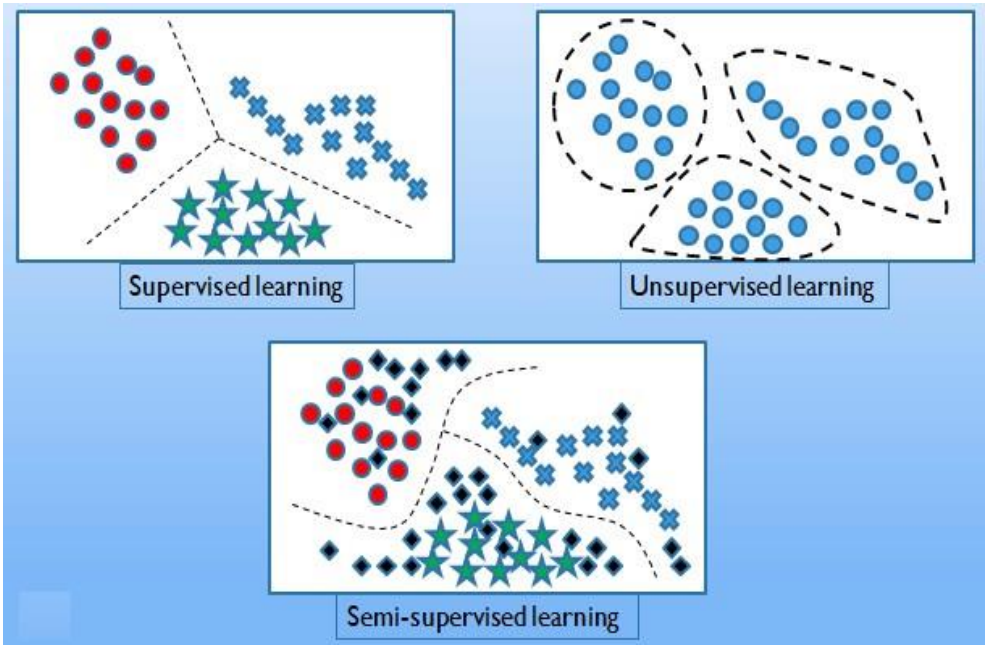


Fig 1.2 Types of Learning

Hypothesis Space

- One way to think about a supervised learning machine is as a device that explores a "hypothesis space".
  - Each setting of the parameters in the machine is a different hypothesis about the function that maps input vectors to output vectors.
  - If the data is noise-free, each training example rules out a region of hypothesis space.
  - If the data is noisy, each training example scales the posterior probability of each point in the hypothesis space in proportion to how likely the training example is given that hypothesis.
- The art of supervised machine learning is in:
  - Deciding how to represent the inputs and outputs
  - Selecting a hypothesis space that is powerful enough to represent the relationship between inputs and outputs but simple enough to be searched.

Generalization

- The real aim of supervised learning is to do well on test data that is not known duringlearning.
- Choosing the values for the parameters that minimize the loss function on the trainingdata is not necessarily the best policy.
- We want the learning machine to model the true regularities in the data and to ignore thenoise in the data.
  - But the learning machine does not know which regularities are real and which areaccidental quirks of the particular set of training examples we happen to pick.
- So how can we be sure that the machine will generalize correctly to new data?

Training set, Test set and Validation set

- Divide the total dataset into three subsets:
  - Training data is used for learning the parameters of the model.
  - Validation data is not used of learning but is used for deciding what type of modeland what amount of regularization works best.
  - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.

- We could then re-divide the total dataset to get another unbiased estimate of the true errorrate.

II. Learning Associations

- Basket analysis:

$P(Y \mid X)$ probability that somebody who buys X also buys Y where X and Y are products/services.

Example: P (chips | beer) = 0.7 // 70 percent of customers who buy beer also buy chips.

We may want to make a distinction among customers and toward this,estimate P(Y|X,D) where D is the set of customer attributes, for example,gender, age, marital status, and so on, assuming that we have access to this information. If this is a bookseller instead of a supermarket, products can be books or authors. In the case of a Web portal, items correspond to links to Web pages, and we can estimate the links a user is likely to click and use this information to download such pages in advance for faster access.

Classification:

- Example: Credit scoring
- Differentiating between low-risk and high-risk customers from their income and savings
- Discriminant: IF income > θ1 AND savings > θ2 THEN low-risk ELSE high-risk

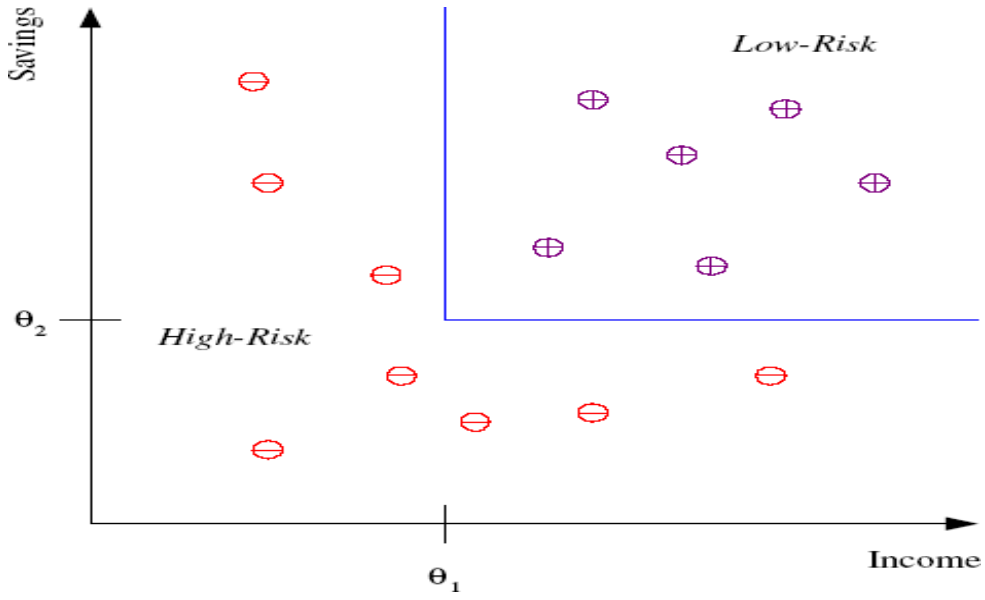


Fig. 1.3 Classification

Prediction - Regression:

- Example: Predict Price of a used car
- x : car attributes

  y : price

$$y = g\,(x \mid \theta\,)$$

where, $g\,(\,)$ is the model, θ are the parameters

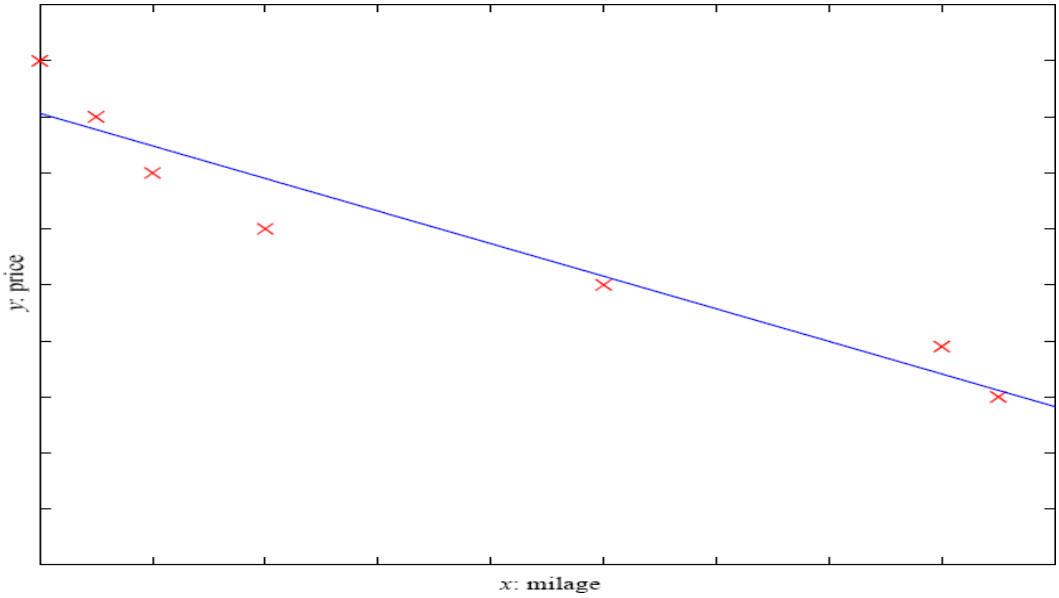


Fig. 1.4 Price Prediction

A training dataset of used cars and the function fitted. For simplicity, mileage is taken as the only input attribute and a linear model is used.

Supervised Learning:

Learning a Class from Examples:

- Class C of a "family car"
  - Prediction: Is car x a family car?
  - Knowledge extraction: What do people expect from a family car?
- Output:
  - Positive (+) and negative (–) examples

- Input representation:

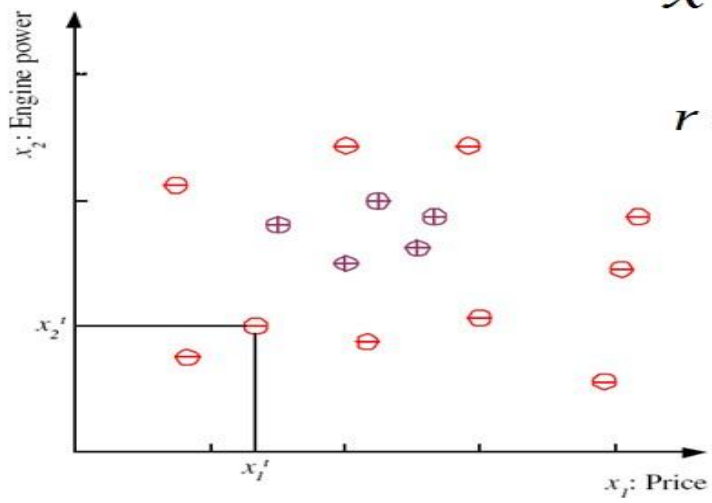
  x_1: price, x_2 : engine power

Training set X

$$\mathcal{X} = \{\boldsymbol{x}^t, r^t\}_{t=1}^N$$

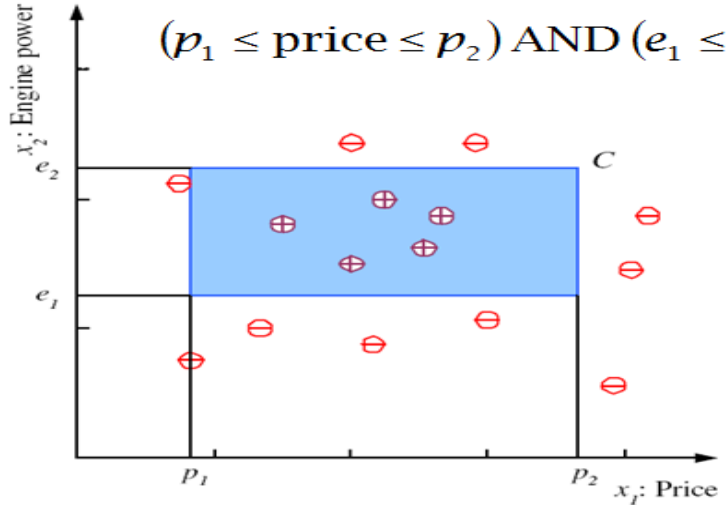
$$r = \begin{cases} 1 \text{ if } \boldsymbol{x} \text{ is positive} \\ 0 \text{ if } \boldsymbol{x} \text{ is negative} \end{cases}$$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



Class C

$$(p_1 \le \text{price} \le p_2) \text{ AND } (e_1 \le \text{engine power} \le e_2)$$



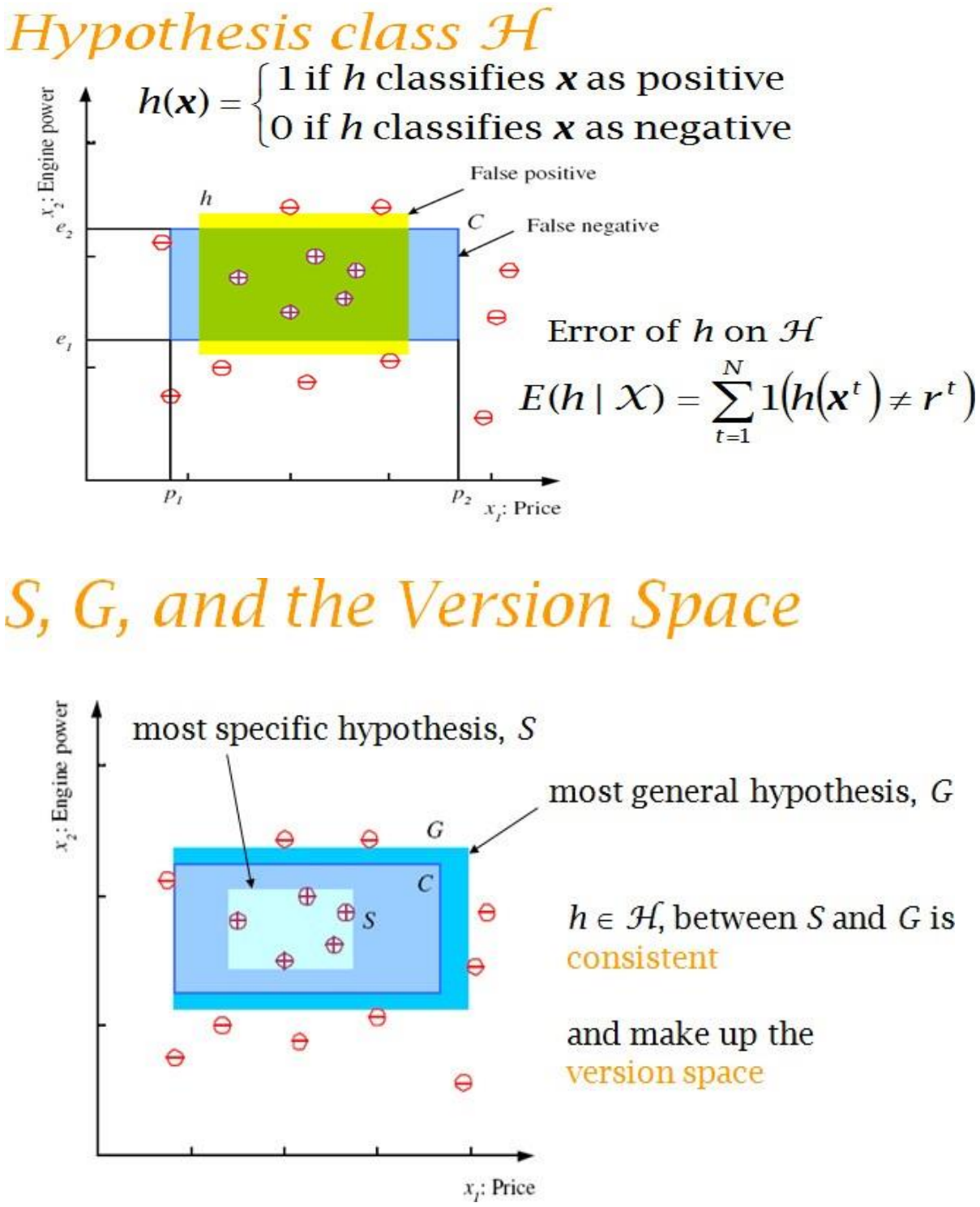


Fig 1.5 Learning a Class from Example

Probably Approximately Correct (PAC):

- Cannot expect a learner to learn a concept exactly.
- Cannot always expect to learn a close approximation to the target concept
- Therefore, the only realistic expectation of a good learner is that with high probability it will learn a close approximation to the target concept.
- In Probably Approximately Correct (PAC) learning, one requires that given small parameters □ and □, with probability at least (1- □) a learner produces a hypothesis with error at most □
- The reason we can hope for that is the Consistent Distribution assumption.

III. PAC Learnability

- Consider a concept class C defined over an instance space X (containing instances oflength n), and a learner L using a hypothesis space H.
- C is PAC learnable by L using H if
    - o for all f □ C,
    - o for all distributions D over X, and fixed 0< □, □ < 1,
- L, given a collection of m examples sampled independently according to D produces
    - o with probability at least (1- □) a hypothesis h □ H with error at most □, (ErrorD =PrD[f(x) : = h(x)])
- where m is polynomial in 1/ □, 1/ □, n and size(H)
- C is efficiently learnable if L can produce the hypothesis in time polynomial in 1/ □, 1/ □,n and size(H)
- We impose two limitations:
- Polynomial sample complexity (information theoretic constraint)
    - o Is there enough information in the sample to distinguish a hypothesis h that approximate f ?
- Polynomial time complexity (computational complexity)
    - o Is there an efficient algorithm that can process the sample and produce a goodhypothesis h ?

- To be PAC learnable, there must be a hypothesis h □ H with arbitrary small error forevery f □ C. We generally assume H □ C. (Properly PAC learnable if H=C)

Occam's Razor:

Claim: The probability that there exists a hypothesis h ∈ H that
(1) is consistent with m examples and
(2) satisfies error(h) > ε ($Error_D(h) = Pr_{x \in D}[f(x) \neg = h(x)]$)
is less than $|H|(1-\varepsilon)^m$.

Proof: Let h be such a bad hypothesis.

- The probability that h is consistent with one example of f is

$$\Pr_{x \in D}[f(x) = h(x)] < 1 - \varepsilon$$

- Since the m examples are drawn independently of each other,
The probability that h is consistent with m example of f is less than $(1-\varepsilon)^m$

- The probability that *some* hypothesis in H is consistent with m examples
is less than $|H|(1-\varepsilon)^m$

We want this probability to be smaller than δ, that is:

$$|H|(1-\varepsilon)^m < \delta$$

$$\ln(|H|) + m \ln(1-\varepsilon) < \ln(\delta)$$

(with $e^{-x} = 1-x+x^2/2+\ldots$; $e^{-x} > 1-x$; $\ln(1-\varepsilon) < -\varepsilon$; gives a safer δ)

$$m > \frac{1}{\varepsilon}\{\ln(|H|) + \ln(1/\delta)\}$$

(gross over estimate)
It is called Occam's razor, because it indicates a preference towards small hypothesis spaces

Noise and Model Complexity:

Noise is any unwanted anomaly in the data and due to noise, the class may be more difficult to learn and zero error may be infeasible with a simple hypothesis class. There are several interpretations of noise:

- There may be imprecision in recording the input attributes, which may shift the data points in the input space.
- There may be errors in labeling the data points, which may relabel positive instances as negative and vice versa. This is sometimes called teacher noise.
- There may be additional attributes, which we have not taken into account, that affect the label of an instance. Such attributes may be hidden or latent in that they may be unobservable. The effect of these neglected attributes is thus modeled as a random component and is included in "noise."

Use the simpler one because

- Simpler to use (lower computational complexity)
- Easier to train (lower space complexity)
- Easier to explain (more interpretable)
- Generalizes better (lower variance - Occam's razor)

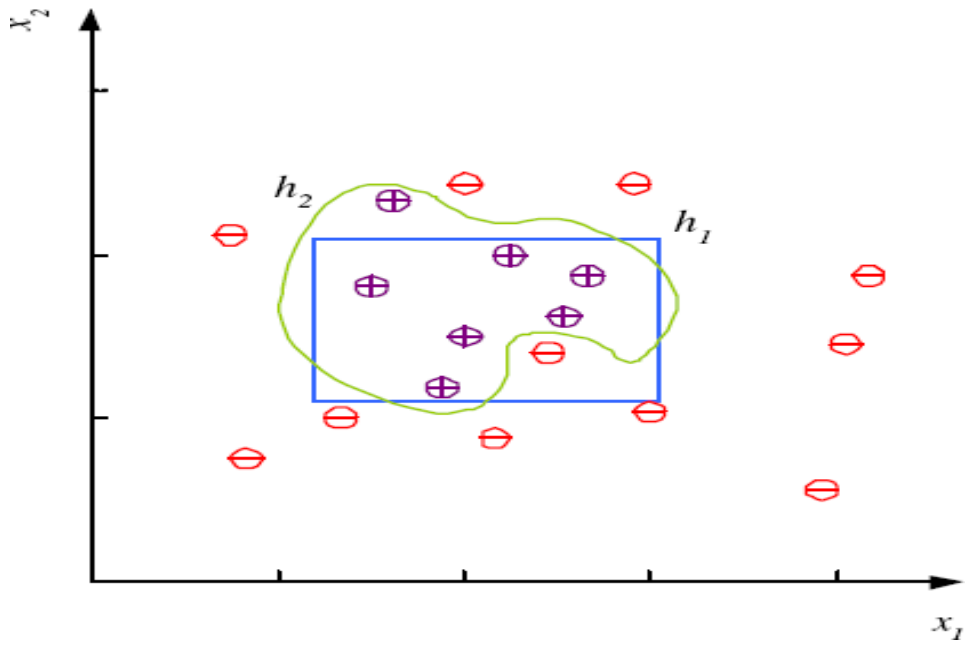


Fig. Model Complexity

Learning Multiple Classes:

In our example of learning a family car, we have positive examples belonging to the class family car and the negative examples belonging to all other cars. This is a *two-class* problem. In the general case, we have *K* classes denoted as C*i*, *i* = 1, . . . , *K*, and an input instance belongs to one and exactly one of them. The training set is now of the form,

$$\mathcal{X} = \{\boldsymbol{x}^t, \boldsymbol{r}^t\}_{t=1}^N$$

$$r_i^t = \begin{cases} 1 \text{ if } \boldsymbol{x}^t \in C_i \\ 0 \text{ if } \boldsymbol{x}^t \in C_j,\ j \neq i \end{cases}$$

Train hypotheses
$h_i(\boldsymbol{x}),\ i = 1, \ldots, K$:

$$h_i(\boldsymbol{x}^t) = \begin{cases} 1 \text{ if } \boldsymbol{x}^t \in C_i \\ 0 \text{ if } \boldsymbol{x}^t \in C_j,\ j \neq i \end{cases}$$

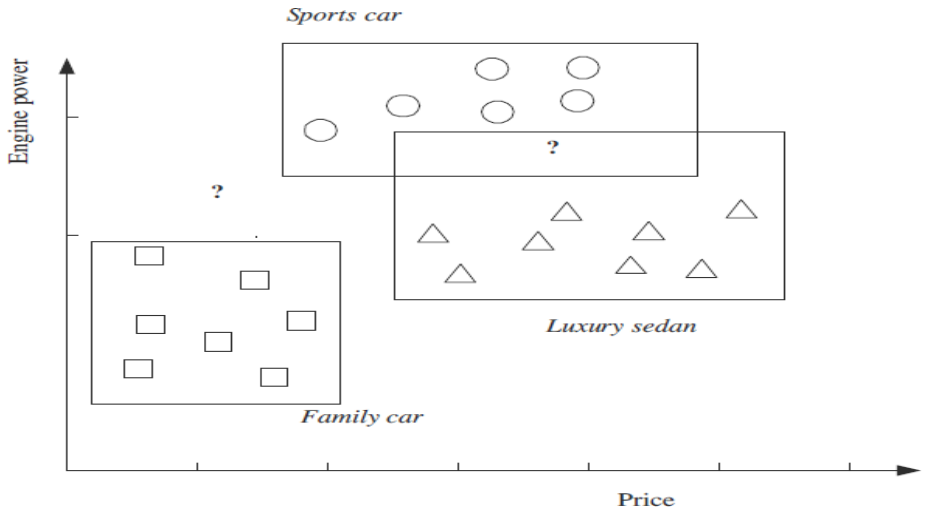


Fig. Learning Multiple Classes

There are three classes: family car, sports car, and luxury sedan. There are three hypotheses induced, each one covering the instances of one class and leaving outside the instances of the other two classes. < ? > are reject regions where no, or more than one, class is chosen.

Model Selection & Generalization:

- Learning is an ill-posed problem; data is not sufficient to find a unique solution
- The need for inductive bias, assumptions about H
- Generalization: How well a model performs on new data
- Overfitting: H more complex than C or f
- Underfitting: H less complex than C or f

Triple Trade-Off:

- There is a trade-off between three factors (Dietterich, 2003):
    1. Complexity of H, *c* (H),
    2. Training set size, *N*,
    3. Generalization error, *E*, on new

        dataAs *N* increases, E decreases

        As *c* (H), first *E* decreases and then *E* increases

Cross-Validation:

- To estimate generalization error, we need data unseen during training. We split the dataas
    - Training set (50%)
    - Validation set (25%)
    - Test (publication) set (25%)
- Resampling when there is few data

Dimensions of a Supervised Learner:

- Model : $g(\boldsymbol{x} \mid \theta)$
- Loss function: $E(\theta \mid \mathcal{X}) = \sum_t L(r^t, g(\boldsymbol{x}^t \mid \theta))$
- Optimization procedure:

$$\theta^* = \arg\min_\theta E(\theta \mid \mathcal{X})$$

UNIT – II

Machine Learning – SCSA1601

UNIT II DECISION THEORY

Bayesian Decision Theory - Introduction - Classification - Discriminant function - Bayesian networks - Association rule - Parametric Methods - Introduction - Estimation - Multivariate Methods - Data Parameter estimation - Dimensionality Reduction - PCA - Linear discriminant analysis.

Bayesian Decision Theory

Bayesian framework assumes that we always have a prior distribution for everything.

- The prior may be very vague.
- When we see some data, we combine our prior distribution with a likelihood term to get a posterior distribution.
- The likelihood term takes into account how probable the observed data is given the parameters of the model.
  - It favors parameter settings that make the data likely.
  - It fights the prior
  - With enough data the likelihood terms always win.

$P(H|\boldsymbol{X})$ is the **posterior probability**, or *a posteriori probability*, of H conditioned on $\boldsymbol{X}$. For example, suppose our world of data tuples is confined to customers described by the attributes *age* and *income*, respectively, and that $\boldsymbol{X}$ is a 35-year-old customer with an income of $40,000. Suppose that H is the hypothesis that our customer will buy a computer. Then $P(H|\boldsymbol{X})$ reflects the probability that customer $\boldsymbol{X}$ will buy a computer given that we know the customer's age and income.

In contrast, $P(H)$ is the **prior probability**, or *a priori probability*, of H. For our example, this is the probability that any given customer will buy a computer, regardless of age, income, or any other information, for that matter.

Bayes' theorem is useful in that it provides a way of calculating the posterior probability $P(H|\boldsymbol{X})$, from $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$.

Bayes' theorem is

$$P(H|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|H)P(H)}{P(\boldsymbol{X})}.$$

Naïve Bayesian Classification

The **naïve Bayesian** classifier, or **simple Bayesian** classifier, works as follows:

1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $X = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, \ldots, A_n$.

2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X. That is, the naïve Bayesian classifier predicts that tuple X belongs to the class C_i if and only if

$$P(C_i|X) > P(C_j|X) \quad \text{for } 1 \leq j \leq m, j \neq i.$$

   Thus, we maximize $P(C_i|X)$. The class C_i for which $P(C_i|X)$ is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Eq. 8.10),

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}.$$

3. As $P(X)$ is constant for all classes, only $P(X|C_i)P(C_i)$ needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \cdots = P(C_m)$, and we would therefore maximize $P(X|C_i)$. Otherwise, we maximize $P(X|C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}|/|D|$, where $|C_{i,D}|$ is the number of training tuples of class C_i in D.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute $P(X|C_i)$. To reduce computation in evaluating $P(X|C_i)$, the naïve assumption of **class-conditional independence** is made. This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$\begin{aligned} P(X|C_i) &= \prod_{k=1}^{n} P(x_k|C_i) \\ &= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i). \end{aligned}$$

   We can easily estimate the probabilities $P(x_1|C_i), P(x_2|C_i), \ldots, P(x_n|C_i)$ from the training tuples. Recall that here x_k refers to the value of attribute A_k for tuple X. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute $P(X|C_i)$, we consider the following:

(a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k, divided by $|C_{i,D}|$, the number of tuples of class C_i in D.

(b) If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation σ, defined by

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

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}).$$

Given database:

Class-Labeled Training Tuples from the *AllElectronics* Customer Database

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Example:

Predicting a class label using naïve Bayesian classification. We wish to predict the class label of a tuple using naïve Bayesian classification, given the same training data as in Example 8.3 for decision tree induction. The training data were shown earlier in Table 8.1. The data tuples are described by the attributes *age*, *income*, *student*, and *credit_rating*. The class label attribute, *buys_computer*, has two distinct values (namely, {*yes*, *no*}). Let C_1 correspond to the class *buys_computer* = *yes* and C_2 correspond to *buys_computer* = *no*. The tuple we wish to classify is

$$\boldsymbol{X} = (age = youth,\ income = medium,\ student = yes,\ credit\_rating = fair)$$

We need to maximize $P(\boldsymbol{X}|C_i)P(C_i)$, for $i = 1, 2$. $P(C_i)$, the prior probability of each class, can be computed based on the training tuples:

$$P(buys\_computer = yes) = 9/14 = 0.643$$
$$P(buys\_computer = no) = 5/14 = 0.357$$

To compute $P(\boldsymbol{X}|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

$$P(age = youth \mid buys\_computer = yes) = 2/9 = 0.222$$
$$P(age = youth \mid buys\_computer = no) = 3/5 = 0.600$$
$$P(income = medium \mid buys\_computer = yes) = 4/9 = 0.444$$
$$P(income = medium \mid buys\_computer = no) = 2/5 = 0.400$$
$$P(student = yes \mid buys\_computer = yes) = 6/9 = 0.667$$
$$P(student = yes \mid buys\_computer = no) = 1/5 = 0.200$$
$$P(credit\_rating = fair \mid buys\_computer = yes) = 6/9 = 0.667$$
$$P(credit\_rating = fair \mid buys\_computer = no) = 2/5 = 0.400$$

Using these probabilities, we obtain

$$\begin{aligned} P(\boldsymbol{X}|buys\_computer = yes) &= P(age = youth \mid buys\_computer = yes) \\ &\quad \times P(income = medium \mid buys\_computer = yes) \\ &\quad \times P(student = yes \mid buys\_computer = yes) \\ &\quad \times P(credit\_rating = fair \mid buys\_computer = yes) \\ &= 0.222 \times 0.444 \times 0.667 \times 0.667 = 0.044. \end{aligned}$$

Similarly,

$$P(\boldsymbol{X}|buys\_computer = no) = 0.600 \times 0.400 \times 0.200 \times 0.400 = 0.019.$$

To find the class, C_i, that maximizes $P(\boldsymbol{X}|C_i)P(C_i)$, we compute

$$P(\boldsymbol{X}|buys\_computer = yes)P(buys\_computer = yes) = 0.044 \times 0.643 = 0.028$$
$$P(\boldsymbol{X}|buys\_computer = no)P(buys\_computer = no) = 0.019 \times 0.357 = 0.007$$

Therefore, the naïve Bayesian classifier predicts *buys_computer* = *yes* for tuple $\boldsymbol{X}$.

Losses and Risks:

- Actions: α_i
- Loss of α_i when the state is C_k : λ_{ik}
- Expected risk (Duda and Hart, 1973)

$$R(\alpha_i \mid \boldsymbol{x}) = \sum_{k=1}^{K} \lambda_{ik} P(C_k \mid \boldsymbol{x})$$

$$\text{choose } \alpha_i \text{ if } R(\alpha_i \mid \boldsymbol{x}) = \min_k R(\alpha_k \mid \boldsymbol{x})$$

0/1 Loss

$$\lambda_{ik} = \begin{cases} 0 \text{ if } i = k \\ 1 \text{ if } i \neq k \end{cases}$$

$$\begin{aligned} R(\alpha_i \mid \boldsymbol{x}) &= \sum_{k=1}^{K} \lambda_{ik} P(C_k \mid \boldsymbol{x}) \\ &= \sum_{k \neq i} P(C_k \mid \boldsymbol{x}) \\ &= 1 - P(C_i \mid \boldsymbol{x}) \end{aligned}$$

For minimum risk, choose the most probable class

Reject

$$\lambda_{ik} = \begin{cases} 0 & \text{if } i = k \\ \lambda & \text{if } i = K+1, \quad 0 < \lambda < 1 \\ 1 & \text{otherwise} \end{cases}$$

$$R(\alpha_{K+1} \mid \boldsymbol{x}) = \sum_{k=1}^{K} \lambda P(C_k \mid \boldsymbol{x}) = \lambda$$

$$R(\alpha_i \mid \boldsymbol{x}) = \sum_{k \neq i} P(C_k \mid \boldsymbol{x}) = 1 - P(C_i \mid \boldsymbol{x})$$

$$\begin{aligned} &\text{choose } C_i \quad \text{if } P(C_i \mid \boldsymbol{x}) > P(C_k \mid \boldsymbol{x}) \ \forall k \neq i \text{ and } P(C_i \mid \boldsymbol{x}) > 1 - \lambda \\ &\text{reject} \quad \text{otherwise} \end{aligned}$$

Discriminant Functions

Classification can also be seen as implementing a set of *discriminant f*unctions,

$$\text{choose } C_i \text{ if } g_i(\boldsymbol{x}) = \max_k g_k(\boldsymbol{x})$$

$$g_i(\boldsymbol{x}) = \begin{cases} -R(\alpha_i \mid \boldsymbol{x}) \\ P(C_i \mid \boldsymbol{x}) \\ p(\boldsymbol{x} \mid C_i)P(C_i) \end{cases}$$

$$K \text{ decision regions} \mathcal{R}_1,\ldots,\mathcal{R}_K$$

$$\mathcal{R}_i = \{\boldsymbol{x} \mid g_i(\boldsymbol{x}) = \max_k g_k(\boldsymbol{x})\}$$

*gi(**x**), i* = 1, . . *K*, such that we

- Dichotomizer ($K=2$) vs Polychotomizer ($K>2$)
- $g(\boldsymbol{x}) = g_1(\boldsymbol{x}) - g_2(\boldsymbol{x})$

$$\text{choose} \begin{cases} C_1 \text{ if } g(\boldsymbol{x}) > 0 \\ C_2 \text{ otherwise} \end{cases}$$

- *Log odds:*

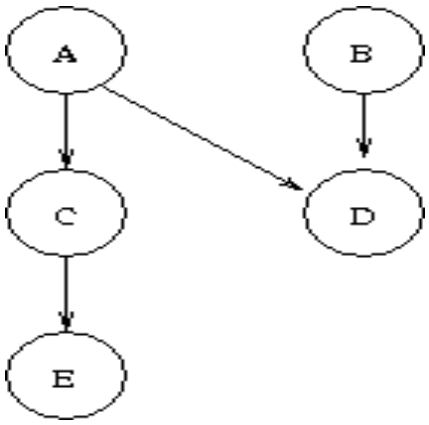
$$\log \frac{P(C_1 \mid \boldsymbol{x})}{P(C_2 \mid \boldsymbol{x})}$$

Bayesian networks

A Bayesian network, Bayes network, belief network, Bayes(ian) model or probabilistic directed acyclic graphical model is a probabilistic graphical model (a type of statistical model) that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). For example, a Bayesian network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases.

Bayesian Net Example:

Consider the following Bayesian network:



Thus, the independence expressed in this Bayesian net are thatA and B are (absolutely) independent.

C is independent of B given A.

D is independent of C given A and B.

E is independent of A, B, and D given C.

Suppose that the net further records the following probabilities:Prob(A=T) = 0.3

Prob(B=T) = 0.6

Prob(C=T|A=T) = 0.8

Prob(C=T|A=F) = 0.4

Prob(D=T|A=T,B=T) = 0.7

Prob(D=T|A=T,B=F) = 0.8

Prob(D=T|A=F,B=T) = 0.1

Prob(D=T|A=F,B=F) = 0.2

Prob(E=T|C=T) = 0.7

Prob(E=T|C=F) = 0.2

Some sample computations:

Prob(D=T):

P(D=T) =

P(D=T,A=T,B=T) + P(D=T,A=T,B=F) + P(D=T,A=F,B=T) + P(D=T,A=F,B=F) =

P(D=T|A=T,B=T) P(A=T,B=T) + P(D=T|A=T,B=F) P(A=T,B=F) +P(D=T|A=F,B=T) P(A=F,B=T) + P(D=T|A=F,B=F) P(A=F,B=F) =

(since A and B are independent absolutely)

P(D=T|A=T,B=T) P(A=T) P(B=T) + P(D=T|A=T,B=F) P(A=T) P(B=F) +P(D=T|A=F,B=T) P(A=F) P(B=T) + P(D=T|A=F,B=F) P(A=F) P(B=F) =

0.7*0.3*0.6 + 0.8*0.3*0.4 + 0.1*0.7*0.6 + 0.2*0.7*0.4 = 0.32

Prob(A=T|C=T):

P(A=T|C=T) = P(C=T|A=T)P(A=T) / P(C=T).

Now, P(C=T) = P(C=T,A=T) + P(C=T,A=F) =
P(C=T|A=T)P(A=T) + P(C=T|A=F)P(A=F) =
0.8*0.3+ 0.4*0.7 = 0.52

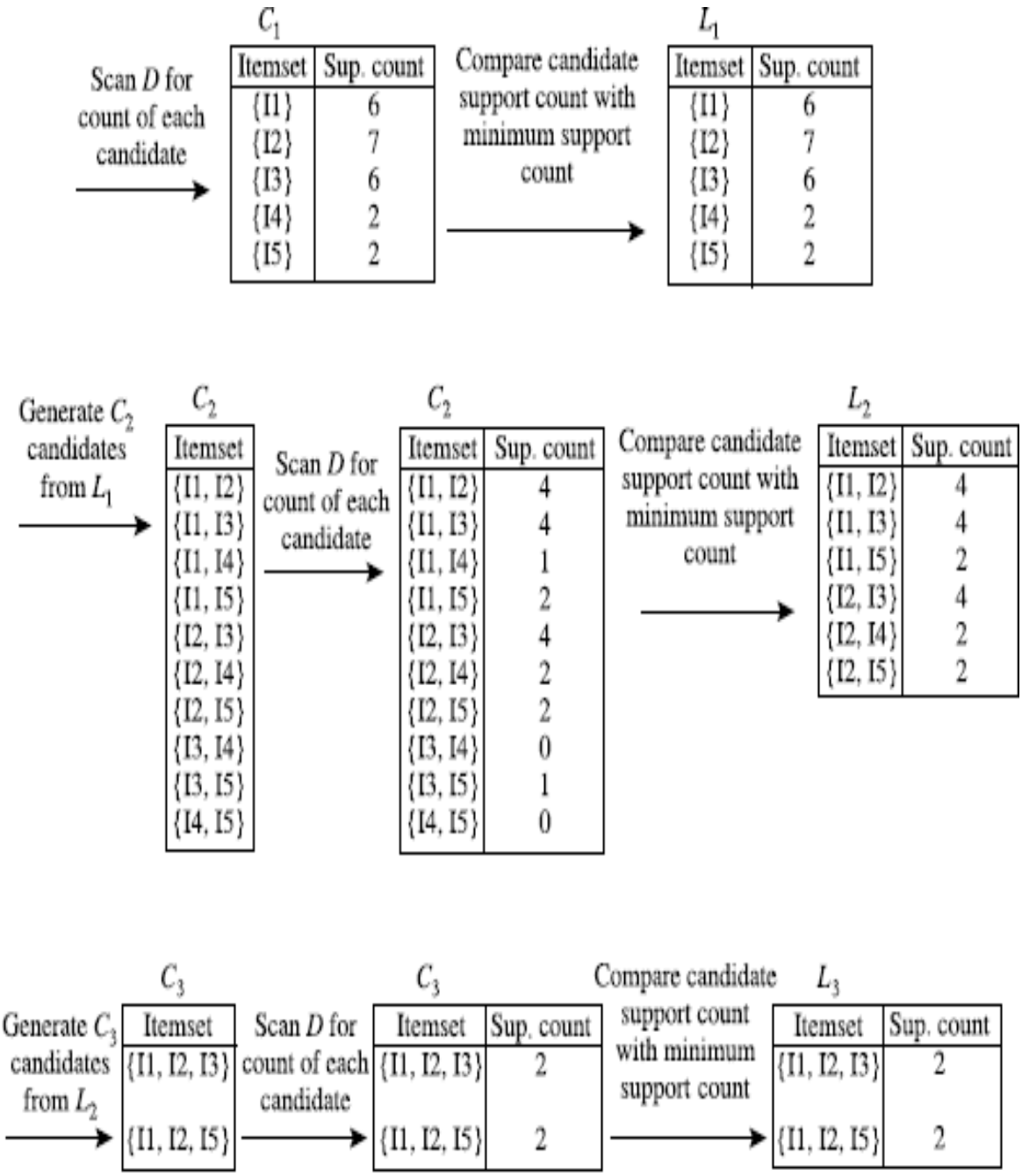
So P(C=T|A=T)P(A=T) / P(C=T) = 0.8*0.3/0.52= 0.46.

Association rule

Association rule mining is explained using the **Apriori Algorithm**.

Transactional Data for an *AllElectronics* Branch

TID	*List of item_IDs*
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3



Generation of the candidate itemsets and frequent itemsets, where the minimum support count is 2.

Apriori Algorithm:

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck | c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

Confidence:

$$confidence(A \Rightarrow B) = P(B|A) = \frac{support\_count(A \cup B)}{support\_count(A)}.$$

The conditional probability is expressed in terms of itemset support count, where $support\_count(A \cup B)$ is the number of transactions containing the itemsets $A \cup B$, and $support\_count(A)$ is the number of transactions containing the itemset A.

Generating association rules. Let's try an example based on the transactional data for *AllElectronics* shown before in Table 6.1. The data contain frequent itemset $X = \{I1, I2, I5\}$. What are the association rules that can be generated from X? The nonempty subsets of X are {I1, I2}, {I1, I5}, {I2, I5}, {I1}, {I2}, and {I5}. The resulting association rules are as shown below, each listed with its confidence:

$\{I1, I2\} \Rightarrow I5$, *confidence* $= 2/4 = 50\%$
$\{I1, I5\} \Rightarrow I2$, *confidence* $= 2/2 = 100\%$
$\{I2, I5\} \Rightarrow I1$, *confidence* $= 2/2 = 100\%$
$I1 \Rightarrow \{I2, I5\}$, *confidence* $= 2/6 = 33\%$
$I2 \Rightarrow \{I1, I5\}$, *confidence* $= 2/7 = 29\%$
$I5 \Rightarrow \{I1, I2\}$, *confidence* $= 2/2 = 100\%$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules are output, because these are the only ones generated that are strong.

Parametric Methods

Parametric Estimation

- X = $\{ x^t \}_t$ where $x^t \sim p\ (x)$
- Parametric estimation:

Assume a form for $p\ (x \mid \vartheta)$ and estimate ϑ, its sufficient statistics, using X

e.g., N (μ, σ^2) where $\vartheta = \{ \mu, \sigma^2 \}$

Maximum Likelihood Estimation:

- Likelihood of θ given the sample X

$$l(\vartheta|X) = p(X|\vartheta) = \prod_t p(x^t|\vartheta)$$

- Log likelihood

$$L(\vartheta|X) = \log l(\vartheta|X) = \sum_t \log p(x^t|\vartheta)$$

- Maximum likelihood estimator (MLE)

$$\vartheta^* = \text{argmax}_\vartheta L(\vartheta|X)$$

Examples: Bernoulli/Multinomial:

- Bernoulli: Two states, failure/success, x in $\{0,1\}$

$$P(x) = p_o^x (1 - p_o)^{(1-x)}$$

$$\mathcal{L}(p_o|\mathcal{X}) = \log \Pi_t p_o^{x^t} (1 - p_o)^{(1-x^t)}$$

MLE: $p_o = \Sigma_t x^t / N$

- Multinomial: $K>2$ states, x_i in $\{0,1\}$

$$P(x_1,x_2,...,x_K) = \Pi_i p_i^{x_i}$$

$$\mathcal{L}(p_1,p_2,...,p_K|\mathcal{X}) = \log \Pi_t \Pi_i p_i^{x_i^t}$$

MLE: $p_i = \Sigma_t x_i^t / N$

Gaussian (Normal) Distribution:

- $p(x) = \mathcal{N}(\mu, \sigma^2)$

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

- MLE for μ and σ^2:

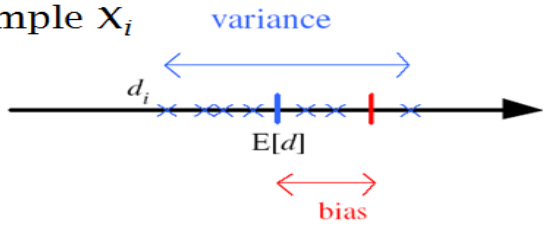
$$m = \frac{\sum_t x^t}{N}$$

$$s^2 = \frac{\sum_t (x^t - m)^2}{N}$$

Bias and Variance:

Unknown parameter θ
Estimator $d_i = d(X_i)$ on sample X_i

Bias: $b_\theta(d) = E[d] - \theta$
Variance: $E[(d-E[d])^2]$



Mean square error:
$r(d,\theta) = E[(d-\theta)^2]$
$= (E[d] - \theta)^2 + E[(d-E[d])^2]$
$= \text{Bias}^2 + \text{Variance}$

Classification

$$g_i(x) = p(x \mid C_i)P(C_i)$$
or equivalently
$$g_i(x) = \log p(x \mid C_i) + \log P(C_i)$$

$$p(x \mid C_i) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left[-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right]$$

$$g_i(x) = -\frac{1}{2}\log 2\pi - \log \sigma_i - \frac{(x-\mu_i)^2}{2\sigma_i^2} + \log P(C_i)$$

- Given the sample $\mathcal{X} = \{x^t, r^t\}_{t=1}^N$

$$x \in \Re \qquad r_i^t = \begin{cases} 1 \text{ if } x^t \in C_i \\ 0 \text{ if } x^t \in C_j, j \neq i \end{cases}$$

- ML estimates are

$$\hat{P}(C_i) = \frac{\sum_t r_i^t}{N} \quad m_i = \frac{\sum_t x^t r_i^t}{\sum_t r_i^t} \quad s_i^2 = \frac{\sum_t (x^t - m_i)^2 r_i^t}{\sum_t r_i^t}$$

- Discriminant becomes

$$g_i(x) = -\frac{1}{2}\log 2\pi - \log s_i - \frac{(x-m_i)^2}{2s_i^2} + \log \hat{P}(C_i)$$

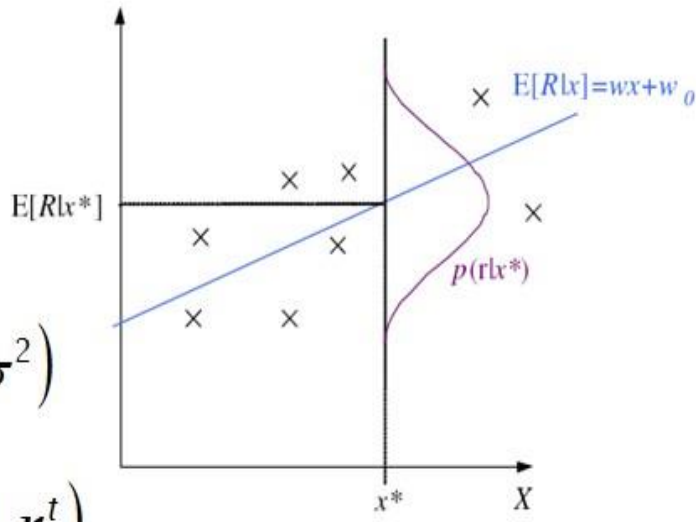
Regression

$$r = f(x) + \varepsilon$$

$$\text{estimator}: g(x \mid \theta)$$

$$\varepsilon \sim \mathcal{N}(0, \sigma^2)$$

$$p(r \mid x) \sim \mathcal{N}(g(x \mid \theta), \sigma^2)$$



$$\mathcal{L}(\theta \mid \mathcal{X}) = \log \prod_{t=1}^{N} p(x^t, r^t)$$

$$= \log \prod_{t=1}^{N} p(r^t \mid x^t) + \log \prod_{t=1}^{N} p(x^t)$$

$$\mathcal{L}(\theta \mid \mathcal{X}) = \log \prod_{t=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{\left[r^t - g(x^t \mid \theta)\right]^2}{2\sigma^2} \right]$$

$$= -N \log \sqrt{2\pi}\sigma - \frac{1}{2\sigma^2} \sum_{t=1}^{N} \left[r^t - g(x^t \mid \theta)\right]^2$$

$$E(\theta \mid \mathcal{X}) = \frac{1}{2} \sum_{t=1}^{N} \left[r^t - g(x^t \mid \theta)\right]^2$$

Linear Regression:

$$g(x^t \mid w_1, w_0) = w_1 x^t + w_0$$

$$\sum_t r^t = N w_0 + w_1 \sum_t x^t$$

$$\sum_t r^t x^t = w_0 \sum_t x^t + w_1 \sum_t (x^t)^2$$

$$\mathbf{A} = \begin{bmatrix} N & \sum_t x^t \\ \sum_t x^t & \sum_t (x^t)^2 \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} \sum_t r^t \\ \sum_t r^t x^t \end{bmatrix}$$

$$\mathbf{w} = \mathbf{A}^{-1}\mathbf{y}$$

Other Error Measures:

- Square Error: $E(\theta \mid \mathcal{X}) = \frac{1}{2}\sum_{t=1}^{N}\left[r^t - g(x^t \mid \theta)\right]^2$
- Relative Square Error: $E(\theta \mid \mathcal{X}) = \frac{\sum_{t=1}^{N}\left[r^t - g(x^t \mid \theta)\right]^2}{\sum_{t=1}^{N}\left[r^t - \bar{r}\right]^2}$
- Absolute Error: $E(\theta|\mathcal{X}) = \sum_t |r^t - g(x^t|\theta)|$
- ε-sensitive Error:

$$E(\theta|\mathcal{X}) = \sum_t 1(|r^t - g(x^t|\theta)|>\varepsilon)\ (|r^t - g(x^t|\theta)| - \varepsilon)$$

Multivariate Methods

Data:

- Multiple measurements (sensors)
- d inputs/features/attributes: d-variate
- N instances/observations/examples

$$\mathbf{X} = \begin{bmatrix} X_1^1 & X_2^1 & \cdots & X_d^1 \\ X_1^2 & X_2^2 & \cdots & X_d^2 \\ \vdots & & & \\ X_1^N & X_2^N & \cdots & X_d^N \end{bmatrix}$$

Multivariate Parameters:

$$\text{Mean} : E[\boldsymbol{x}] = \boldsymbol{\mu} = [\mu_1, ..., \mu_d]^T$$

$$\text{Covariance} : \sigma_{ij} \equiv \text{Cov}(X_i, X_j)$$

$$\text{Correlation} : \text{Corr}(X_i, X_j) \equiv \rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$$

$$\Sigma \equiv \text{Cov}(\boldsymbol{X}) = E\left[(\boldsymbol{X} - \mu)(\boldsymbol{X} - \mu)^T\right] = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1d} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2d} \\ \vdots & & & \\ \sigma_{d1} & \sigma_{d2} & \cdots & \sigma_d^2 \end{bmatrix}$$

Parameter Estimation

Sample mean $\mathbf{m}$: $m_i = \frac{\sum_{t=1}^{N} x_i^t}{N}, i = 1,...,d$

Covariance matrix $\mathbf{S}$: $s_{ij} = \frac{\sum_{t=1}^{N} (x_i^t - m_i)(x_j^t - m_j)}{N}$

Correlation matrix $\mathbf{R}$: $r_{ij} = \frac{s_{ij}}{s_i s_j}$

Classification

- If $p(\mathbf{x} \mid C_i) \sim \mathcal{N}(\mu_i, \Sigma_i)$

$$p(\mathbf{x} \mid C_i) = \frac{1}{(2\pi)^{d/2} |\Sigma_i|^{1/2}} \exp\left[-\frac{1}{2} (\mathbf{x} - \mu_i)^T \Sigma_i^{-1} (\mathbf{x} - \mu_i) \right]$$

- Discriminant functions are

$$g_i(\mathbf{x}) = \log p(\mathbf{x} \mid C_i) + \log P(C_i)$$

$$= -\frac{d}{2} \log 2\pi - \frac{1}{2} \log |\Sigma_i| - \frac{1}{2} (\mathbf{x} - \mu_i)^T \Sigma_i^{-1} (\mathbf{x} - \mu_i) + \log P(C_i)$$

Estimating the mean and Variance,

$$\hat{P}(C_i) = \frac{\sum_t r_i^t}{N}$$

$$\mathbf{m}_i = \frac{\sum_t r_i^t \mathbf{x}^t}{\sum_t r_i^t}$$

$$\mathbf{S}_i = \frac{\sum_t r_i^t (\mathbf{x}^t - \mathbf{m}_i)(\mathbf{x}^t - \mathbf{m}_i)^T}{\sum_t r_i^t}$$

$$g_i(\mathbf{x}) = -\frac{1}{2}\log|\mathbf{S}_i| - \frac{1}{2}(\mathbf{x} - \mathbf{m}_i)^T \mathbf{S}_i^{-1}(\mathbf{x} - \mathbf{m}_i) + \log \hat{P}(C_i)$$

<u>Quadratic Discriminant:</u>

$$g_i(\mathbf{x}) = -\frac{1}{2}\log|\mathbf{S}_i| - \frac{1}{2}\left(\mathbf{x}^T \mathbf{S}_i^{-1}\mathbf{x} - 2\mathbf{x}^T \mathbf{S}_i^{-1}\mathbf{m}_i + \mathbf{m}_i^T \mathbf{S}_i^{-1}\mathbf{m}_i\right) + \log \hat{P}(C_i)$$

$$= \mathbf{x}^T \mathbf{W}_i \mathbf{x} + \mathbf{w}_i^T \mathbf{x} + w_{i0}$$

where

$$\mathbf{W}_i = -\frac{1}{2}\mathbf{S}_i^{-1}$$

$$\mathbf{w}_i = \mathbf{S}_i^{-1}\mathbf{m}_i$$

$$w_{i0} = -\frac{1}{2}\mathbf{m}_i^T \mathbf{S}_i^{-1}\mathbf{m}_i - \frac{1}{2}\log|\mathbf{S}_i| + \log \hat{P}(C_i)$$

Tuning Complexity

Assumption	*Covariance matrix*	*No of parameters*
Shared, Hyperspheric	$\mathbf{S}_i=\mathbf{S}=s^2\mathbf{I}$	1
Shared, Axis-aligned	$\mathbf{S}_i=\mathbf{S}$, with $s_{ij}=0$	d
Shared, Hyperellipsoidal	$\mathbf{S}_i=\mathbf{S}$	$d(d+1)/2$
Different, Hyperellipsoidal	$\mathbf{S}_i$	$K\,d(d+1)/2$

- As we increase complexity (less restricted **S**), bias decreases and variance increases
- Assume simple models (allow some bias) to control variance (regularization)

Discrete Features

- Binary features: $p_{ij} \equiv p(x_j = 1 \mid C_i)$

  if x_j are independent (Naive Bayes')

$$p(\mathbf{x} \mid C_i) = \prod_{j=1}^{d} p_{ij}^{x_j} (1 - p_{ij})^{(1-x_j)}$$

  the discriminant is linear

$$g_i(\mathbf{x}) = \log p(\mathbf{x} \mid C_i) + \log P(C_i)$$
$$= \sum_j \left[x_j \log p_{ij} + (1 - x_j) \log (1 - p_{ij}) \right] + \log P(C_i)$$

  Estimated parameters $\hat{p}_{ij} = \dfrac{\sum_t x_j^t r_i^t}{\sum_t r_i^t}$

- Multinomial (1-of-n_j) features: $x_j \in \{v_1, v_2, ..., v_{n_j}\}$

$$p_{ijk} \equiv p(z_{jk} = 1 \mid C_i) = p(x_j = v_k \mid C_i)$$

if x_j are independent

$$p(\mathbf{x} \mid C_i) = \prod_{j=1}^{d} \prod_{k=1}^{n_j} p_{ijk}^{z_{jk}}$$

$$g_i(\mathbf{x}) = \sum_j \sum_k z_{jk} \log p_{ijk} + \log P(C_i)$$

$$\hat{p}_{ijk} = \frac{\sum_t z_{jk}^t r_i^t}{\sum_t r_i^t}$$

Dimensionality Reduction

Necessity:

1. Reduces time complexity: Less computation
2. Reduces space complexity: Less parameters
3. Saves the cost of observing the feature
4. Simpler models are more robust on small datasets
5. More interpretable; simpler explanation
6. Data visualization (structure, groups, outliers, etc) if plotted in 2 or 3 dimensions.

Feature Selection and Extraction:

- Feature selection: Choosing $k<d$ important features, ignoring the remaining $d - k$

  Subset selection algorithms
- Feature extraction: Project the original x_i , $i=1,...,d$ dimensions to new $k<d$ dimensions, z_j , $j=1,...,k$

  Principal components analysis (PCA), linear discriminant analysis (LDA), factor analysis (FA)

Principal Component Analysis(PCA)

- Find a low-dimensional space such that when $\boldsymbol{x}$ is projected there, information loss is minimized.
- The projection of $\boldsymbol{x}$ on the direction of $\boldsymbol{w}$ is: $z = \boldsymbol{w}^T\boldsymbol{x}$
- Find $\boldsymbol{w}$ such that Var(z) is maximized

$$\begin{aligned}\text{Var}(z) &= \text{Var}(\boldsymbol{w}^T\boldsymbol{x}) = E[(\boldsymbol{w}^T\boldsymbol{x} - \boldsymbol{w}^T\boldsymbol{\mu})^2] \\ &= E[(\boldsymbol{w}^T\boldsymbol{x} - \boldsymbol{w}^T\boldsymbol{\mu})(\boldsymbol{w}^T\boldsymbol{x} - \boldsymbol{w}^T\boldsymbol{\mu})] \\ &= E[\boldsymbol{w}^T(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T\boldsymbol{w}] \\ &= \boldsymbol{w}^T E[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T]\boldsymbol{w} = \boldsymbol{w}^T\boldsymbol{\Sigma}\,\boldsymbol{w}\end{aligned}$$

where $\text{Var}(\boldsymbol{x}) = E[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T] = \boldsymbol{\Sigma}$

- Maximize Var(z) subject to $\|\boldsymbol{w}\|=1$

$$\max_{\boldsymbol{w}_1} \boldsymbol{w}_1^T \boldsymbol{\Sigma} \boldsymbol{w}_1 - \alpha\left(\boldsymbol{w}_1^T \boldsymbol{w}_1 - 1\right)$$

$\boldsymbol{\Sigma w}_1 = \alpha \boldsymbol{w}_1$ that is, $\boldsymbol{w}_1$ is an eigenvector of $\boldsymbol{\Sigma}$

Choose the one with the largest eigenvalue for Var(z) to be max

- Second principal component: Max Var(z_2), s.t., $\|\boldsymbol{w}_2\|=1$ and orthogonal to $\boldsymbol{w}_1$

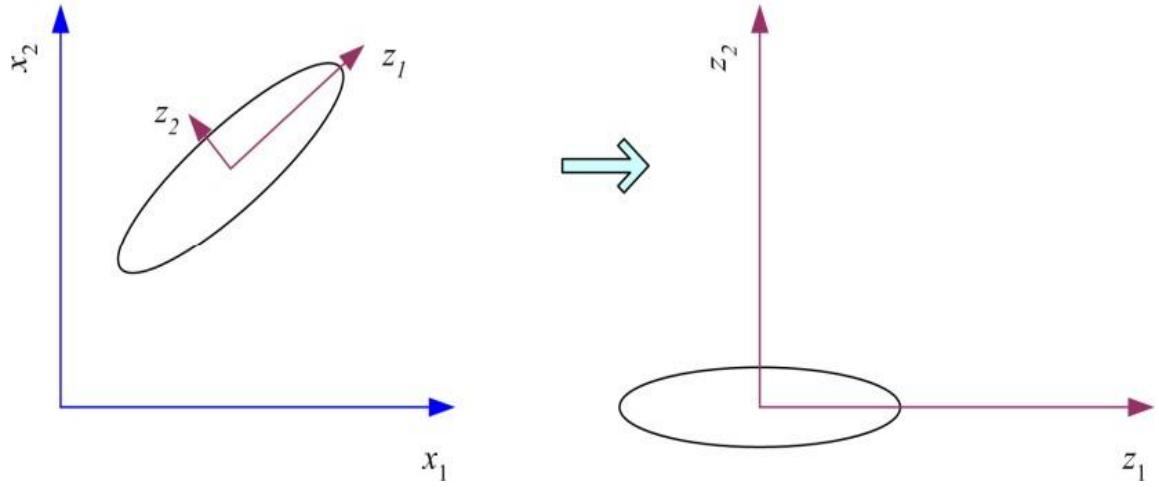
$$\max_{\boldsymbol{w}_2} \boldsymbol{w}_2^T \boldsymbol{\Sigma} \boldsymbol{w}_2 - \alpha\left(\boldsymbol{w}_2^T \boldsymbol{w}_2 - 1\right) - \beta\left(\boldsymbol{w}_2^T \boldsymbol{w}_1 - 0\right)$$

$\boldsymbol{\Sigma}\, \boldsymbol{w}_2 = \alpha\, \boldsymbol{w}_2$ that is, $\boldsymbol{w}_2$ is another eigenvector of $\boldsymbol{\Sigma}$ and so on.

$$\boldsymbol{z} = \mathbf{W}^T(\boldsymbol{x} - \boldsymbol{m})$$

where the columns of $\mathbf{W}$ are the eigenvectors of $\boldsymbol{\Sigma}$, and $\boldsymbol{m}$ is sample mean

Centers the data at the origin and rotates the axes



- Proportion of Variance (PoV) explained

$$\frac{\lambda_1 + \lambda_2 + \cdots + \lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_k + \cdots + \lambda_d}$$

when λ_i are sorted in descending order
- Typically, stop at PoV>0.9
- Scree graph plots of PoV vs k, stop at "elbow"

Factor Analysis

- Find a small number of factors $\mathbf{z}$, which when combined generate $\mathbf{x}$:

$$x_i - \mu_i = v_{i1}z_1 + v_{i2}z_2 + \ldots + v_{ik}z_k + \varepsilon_i$$

where z_j, $j=1,\ldots,k$ are the latent factors with

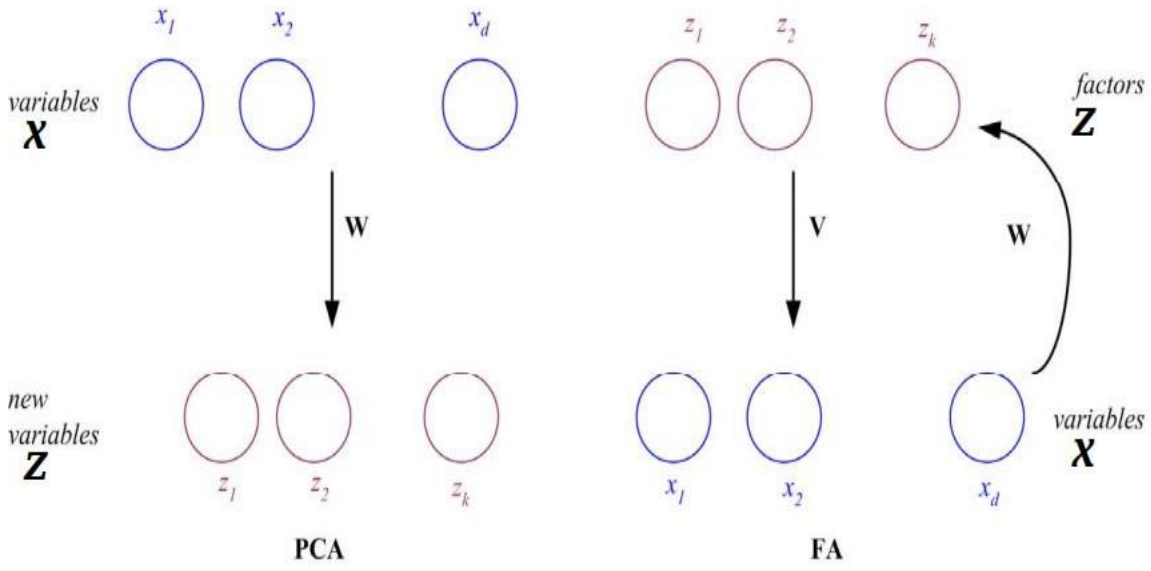
$$E[z_j]=0,\ \mathrm{Var}(z_j)=1,\ \mathrm{Cov}(z_i, z_j)=0,\ i \neq j,$$

ε_i are the noise sources

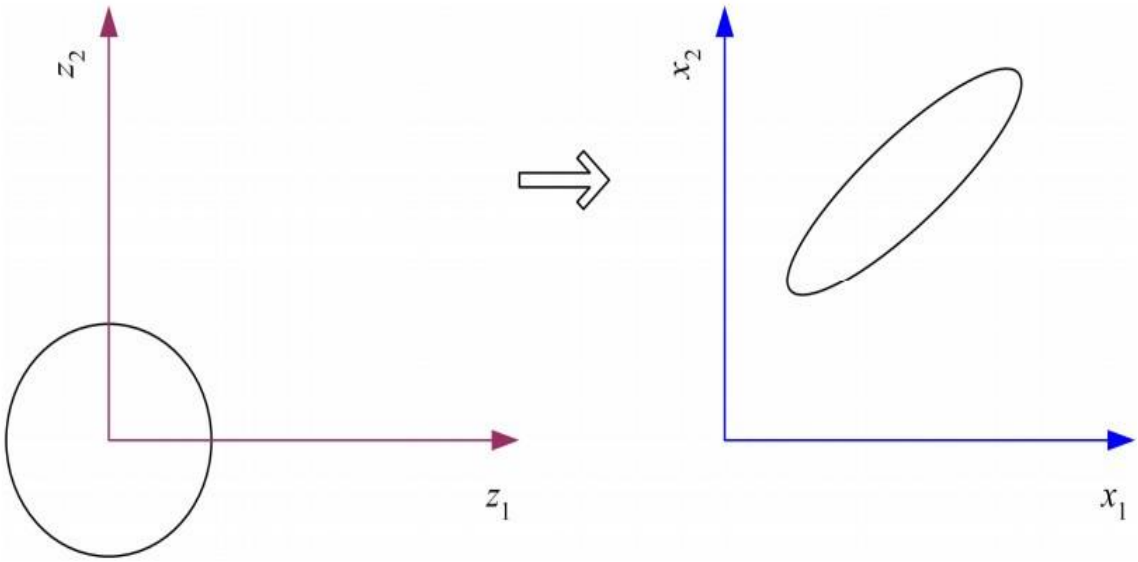
$$E[\varepsilon_i]=\psi_i,\ \mathrm{Cov}(\varepsilon_i, \varepsilon_j)=0,\ i \neq j,\ \mathrm{Cov}(\varepsilon_i, z_j)=0,$$

and v_{ij} are the factor loadings

- PCA From $\boldsymbol{x}$ to $\boldsymbol{z}$ $\boldsymbol{z} = \mathbf{W}^T(\boldsymbol{x} - \boldsymbol{\mu})$
- FA From $\boldsymbol{z}$ to $\boldsymbol{x}$ $\boldsymbol{x} - \boldsymbol{\mu} = \mathbf{V}\boldsymbol{z} + \boldsymbol{\varepsilon}$



- In FA, factors z_j are stretched, rotated and translated to generate $\boldsymbol{x}$



Multidimensional Scaling

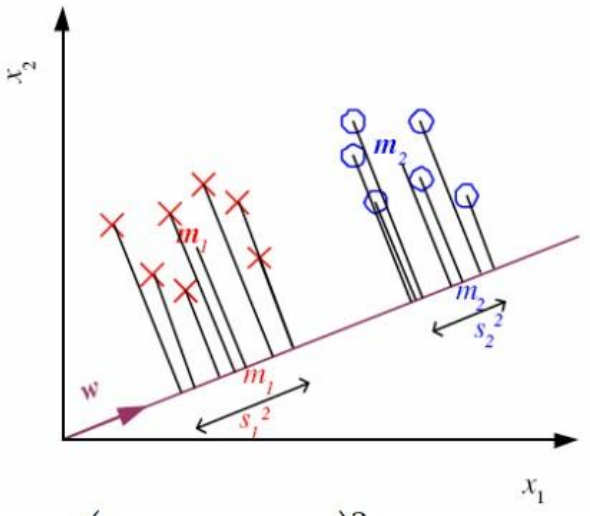
- Given pairwise distances between N points,
  d_{ij}, $i,j = 1,\ldots,N$
  place on a low-dim map s.t. distances are preserved.
- $\boldsymbol{z} = \boldsymbol{g}(\boldsymbol{x} \mid \theta)$ Find θ that min Sammon stress

$$E(\theta \mid \mathcal{X}) = \sum_{r,s} \frac{\left(\left\|\boldsymbol{z}^r - \boldsymbol{z}^s\right\| - \left\|\boldsymbol{x}^r - \boldsymbol{x}^s\right\|\right)^2}{\left\|\boldsymbol{x}^r - \boldsymbol{x}^s\right\|^2}$$

$$= \sum_{r,s} \frac{\left(\left\|\boldsymbol{g}(\boldsymbol{x}^r \mid \theta) - \boldsymbol{g}(\boldsymbol{x}^s \mid \theta)\right\| - \left\|\boldsymbol{x}^r - \boldsymbol{x}^s\right\|\right)^2}{\left\|\boldsymbol{x}^r - \boldsymbol{x}^s\right\|^2}$$

Linear Discriminant Analysis

- Find a low-dimensional space such that when $\boldsymbol{x}$ is projected, classes are well-separated.
- Find $\boldsymbol{w}$ that maximizes



$$J(\boldsymbol{w}) = \frac{(m_1 - m_2)^2}{s_1^2 + s_2^2}$$

$$m_1 = \frac{\sum_t \boldsymbol{w}^T \boldsymbol{x}^t r^t}{\sum_t r^t} \qquad s_1^2 = \sum_t \left(\boldsymbol{w}^T \boldsymbol{x}^t - m_1\right)^2 r^t$$

- Between-class scatter:

$$(m_1 - m_2)^2 = \left(\mathbf{w}^T\mathbf{m}_1 - \mathbf{w}^T\mathbf{m}_2\right)^2$$
$$= \mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T\mathbf{w}$$
$$= \mathbf{w}^T\mathbf{S}_B\mathbf{w} \text{ where } \mathbf{S}_B = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T$$

- Within-class scatter:

$$s_1^2 = \sum_t \left(\mathbf{w}^T\mathbf{x}^t - m_1\right)^2 r^t$$
$$= \sum_t \mathbf{w}^T\left(\mathbf{x}^t - \mathbf{m}_1\right)\left(\mathbf{x}^t - \mathbf{m}_1\right)^T\mathbf{w}r^t = \mathbf{w}^T\mathbf{S}_1\mathbf{w}$$

$$\text{where } \mathbf{S}_1 = \sum_t \left(\mathbf{x}^t - \mathbf{m}_1\right)\left(\mathbf{x}^t - \mathbf{m}_1\right)^T r^t$$

$$s_1^2 + s_1^2 = \mathbf{w}^T\mathbf{S}_W\mathbf{w} \text{ where } \mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2$$

Fisher's Linear Discriminant:

- Find $\mathbf{w}$ that max

$$J(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}} = \frac{\left|\mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)\right|^2}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}}$$

- LDA soln:

$$\mathbf{w} = c \cdot \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$$

- Parametric soln:

$$\mathbf{w} = \Sigma^{-1}(\mu_1 - \mu_2)$$
$$\text{when } p(\mathbf{x} \mid C_i) \sim \mathcal{N}(\mu_i, \Sigma)$$

For K>2 Classes,

- Within-class scatter:

$$\mathbf{S}_W = \sum_{i=1}^{K} \mathbf{S}_i \qquad \mathbf{S}_i = \sum_t r_i^t \left(\mathbf{x}^t - \mathbf{m}_i\right)\left(\mathbf{x}^t - \mathbf{m}_i\right)^T$$

- Between-class scatter:

$$\mathbf{S}_B = \sum_{i=1}^{K} N_i \left(\mathbf{m}_i - \mathbf{m}\right)\left(\mathbf{m}_i - \mathbf{m}\right)^T \qquad \mathbf{m} = \frac{1}{K}\sum_{i=1}^{K} \mathbf{m}_i$$

- Find $\mathbf{W}$ that max

$$J(\mathbf{W}) = \frac{\left|\mathbf{W}^T \mathbf{S}_B \mathbf{W}\right|}{\left|\mathbf{W}^T \mathbf{S}_W \mathbf{W}\right|}$$

The largest eigenvectors of $\mathbf{S}_W^{-1}\mathbf{S}_B$
Maximum rank of K-1

UNIT – III

Machine Learning – SCSA1601

UNIT III CLUSTERING & REGRESSION

Clustering - Mixture densities - k-means clustering - Supervised Learning after clustering - Hierarchical clustering - Nonparametric Methods - Density estimation - Generalization of multivariate data - Smoothing models - Decision Trees - Univariate trees - Multivariate trees -Learning rules from data - Linear Discrimination- Gradient Descent.

Clustering

- Cluster: A collection of data objects
  - similar (or related) to one another within the same group
  - dissimilar (or unrelated) to the objects in other groups
- Cluster analysis (or clustering, data segmentation, …)
  - Finding similarities between data according to the characteristics found in the dataand grouping similar data objects into clusters
- Unsupervised learning: no predefined classes (i.e., learning by observations vs. learningby examples: supervised)
- A good clustering method will produce high quality clusters
  - high intra-class similarity: cohesive within clusters
  - low inter-class similarity: distinctive between clusters
- The quality of a clustering method depends on
  - the similarity measure used by the method
  - its implementation, and
  - its ability to discover some or all of the hidden patterns.

Major Clustering Approaches

- Partitioning approach:
  - Construct various partitions and then evaluate them by some criterion, e.g.,minimizing the sum of square errors
  - Typical methods: k-means, k-medoids, CLARANS
- Hierarchical approach:
  - Create a hierarchical decomposition of the set of data (or objects) using somecriterion
  - Typical methods: DIANA, AGNES, BIRCH, CAMELEON

Mixture densities

The *mixture density* is written as

$$p(\boldsymbol{x}) = \sum_{i=1}^{k} p(\boldsymbol{x}|\mathcal{G}_i)P(\mathcal{G}_i)$$

where $\mathcal{G}_i$ are the *mixture components*. They are also called *group* or *clusters*. $p(\boldsymbol{x}|\mathcal{G}_i)$ are the *component densities* and $P(\mathcal{G}_i)$ are the *mixture proportions*. The number of components, k, is a hyperparameter and should be specified beforehand. Given a sample and k, learning corresponds to estimating the component densities and proportions. When we assume that the component densities obey a parametric model, we need only estimate their parameters. If the component densities are multivariate Gaussian, we have $p(\boldsymbol{x}|\mathcal{G}_i) \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, and $\Phi = \{P(\mathcal{G}_i), \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}_{i=1}^{k}$ are the parameters that should be estimated from the iid sample $\mathcal{X} = \{\boldsymbol{x}^t\}_t$.

Parametric classification is a bona fide mixture model where groups, $\mathcal{G}_i$, correspond to classes, C_i, component densities $p(\boldsymbol{x}|\mathcal{G}_i)$ correspond to class densities $p(\boldsymbol{x}|C_i)$, and $P(\mathcal{G}_i)$ correspond to class priors, $P(C_i)$:

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} p(\boldsymbol{x}|C_i)P(C_i)$$

In this *supervised* case, we know how many groups there are and learning the parameters is trivial because we are given the labels, namely, which instance belongs to which class (component). We remember from chapter 5 that when we are given the sample $\mathcal{X} = \{\boldsymbol{x}^t, \boldsymbol{r}^t\}_{t=1}^{N}$, where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_i$ and 0 otherwise, the parameters can be calculated using maximum likelihood. When each class is Gaussian distributed, we have a Gaussian mixture, and the parameters are estimated as

$$\begin{aligned}
\hat{P}(C_i) &= \frac{\sum_t r_i^t}{N} \\
\boldsymbol{m}_i &= \frac{\sum_t r_i^t \boldsymbol{x}^t}{\sum_t r_i^t} \\
\mathbf{S}_i &= \frac{\sum_t r_i^t (\boldsymbol{x}^t - \boldsymbol{m}_i)(\boldsymbol{x}^t - \boldsymbol{m}_i)^T}{\sum_t r_i^t}
\end{aligned}$$

The difference in this chapter is that the sample is $\mathcal{X} = \{\boldsymbol{x}^t\}_t$: We have an *unsupervised learning* problem. We are given only $\boldsymbol{x}^t$ and not the labels $\boldsymbol{r}^t$, that is, we do not know which $\boldsymbol{x}^t$ comes from which component. So we should estimate both: First, we should estimate the labels, r_i^t, the component that a given instance belongs to; and, second, once we estimate the labels, we should estimate the parameters of the components given the set of instances belonging to them. We are first going to discuss a simple algorithm, *k*-means clustering, for this purpose and later on show that it is a special case of the *Expectation-Maximization* algorithm.

Partitioning method

- Partitioning a database *D* of *n* objects into a set of *k* clusters, such that the sum of squareddistances is minimized (where c_i is the centroid or medoid of cluster C_i)
- Given *k*, find a partition of *k clusters* that optimizes the chosen partitioning criterion

The K-Means Clustering Method

- Given *k*, the *k-means* algorithm is implemented in four steps:
  - Partition objects into *k* nonempty subsets
  - Compute seed points as the centroids of the clusters of the current partitioning(the centroid is the center, i.e., *mean point*, of the cluster)
  - Assign each object to the cluster with the nearest seed point
  - Go back to Step 2, stop when the assignment does not change

Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to k random $\boldsymbol{x}^t$
Repeat
  For all $\boldsymbol{x}^t \in \mathcal{X}$
    $$b_i^t \leftarrow \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\| \\ 0 & \text{otherwise} \end{cases}$$
  For all $\boldsymbol{m}_i, i = 1, \ldots, k$
    $$\boldsymbol{m}_i \leftarrow \sum_t b_i^t \boldsymbol{x}^t / \sum_t b_i^t$$
Until $\boldsymbol{m}_i$ converge

k-means algorithm.

Example:

K-Means clustering:-

1) Decide the number & frame the clusters.
2) Find Mean of clusters.
3) Find distance b/w Mean & distance for all points.

Eg:- $x_1 = \{1, 0\}$, $x_2 = \{0, 1\}$, $x_3 = \{2, 1\}$, $x_4 = \{2, 3\}$.

Assume, we take two clusters, @

$C_1 = \{x_1, x_3\}$ & $C_2 = \{x_2, x_4\}$

(a) Apply 1 iteration of K-means partional clustering algorithm

(b) What is change in total square error.

(c) Apply 2nd iteration of K-means algorithm.

step 1:-

Mean,

$$M_k = \frac{1}{n_k} \sum_{i=1}^{k} x_{ik}$$

C_1:-

$x_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

$x_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$

$\Rightarrow M_1 = \left\{ \frac{1+2}{2}, \frac{0+1}{2} \right\}$

$= \{1.5, 0.5\}$.

$M_2 = \left\{ \frac{0+2}{2}, \frac{1+3}{2} \right\}$

$= \{1.5, 2\}$.

C_2 :- $x_3 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$ $x_4 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$

Step 2 :-

Error Calculation.

$$e_k^2 = \sum_{i=1}^{k} (x_{ik} - M_{ik})^2$$

$M_1 = (1.5, 0.5)$ | $M_2 = (1.5, 2)$
$x_1 = (1, 0)$ | $x_3 = (0, 2)$
$x_2 = (2, 1)$ | $x_4 = (2, 3)$

$C_1 \Rightarrow e_1^2 = \left[(1-1.5)^2 + (0-0.5)^2 + (2-1.5)^2 + (1-0.5)^2 \right]$

$= 1.$

$C_2 \Rightarrow e_2^2 = \left[(0-1.5)^2 + (1-2)^2 + (3-1.5)^2 + (2-2)^2 \right]$

$= 6.5$

Step 3 :-

Total square error.

$$\Sigma_k^2 = \sum_{k=1}^{k} e_k^2$$

$\Rightarrow e_1^2 + e_2^2$

$1 + 6.5 = 7.5$

Step 4 :-

Distance calculation.

k-means follow euclidean's distance measure,

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

X_1 (1, 0):-

$d(X_1, M_1) = \sqrt{(1-1.5)^2 + (0-0.5)^2} = 0.707$

$d(X_1, M_2) = \sqrt{(1-1.5)^2 + (0-2)^2} = 2.062$

X_2 (0, 1):-

$d(X_2, M_1) = \sqrt{(0-1.5)^2 + (1-0.5)^2} = 1.581$

$d(X_2, M_2) = \sqrt{(0-1.5)^2 + (1-2)^2} = 1.803$

X_3 (2, 1):-

$d(X_3, M_1) = \sqrt{(2-1.5)^2 + (1-0.5)^2} = 0.707$

$d(X_3, M_2) = \sqrt{(2-1.5)^2 + (1-2)^2} = 1.118$

X_4 (3, 3):-

$d(X_4, M_1) = \sqrt{(3-1.5)^2 + (3-0.5)^2} = 2.915$

$d(X_4, M_2) = \sqrt{(3-1.5)^2 + (3-2)^2} = 1.8027$

	M_1	M_2
X_1	0.707	2.062
X_2	1.581	1.803
X_3	0.707	1.118
X_4	2.915	1.8027
	C_1	C_2

1) check the smallest element row wise

2) Divide the columns C_1 & C_2.

$\Rightarrow C_1 = (X_1, X_2, X_3)$

$C_2 = (X_4)$.

(c)

Mean,

$$M_i = \frac{1}{n_i} \sum_{i=1}^{r} x_{ij}$$

C_1:-

$x_1 = (1, 0)$

$x_2 = (0, 1)$ $\Rightarrow$ $M_1 = \left\{ \frac{1+0+0}{3}, \frac{0+1+1}{3} \right\}$

$x_3 = (2, 1)$ $= (1, 0.66)$

C_2:-

$x_4 = (3, 3)$ $\Rightarrow$ $M_2 = \{3, 3\}$.

Error Calculation,

$C_1 \Rightarrow e_1^2 \Rightarrow [(1-1)^2 + (0-0.66)^2 + (0-1)^2 + (1-0.66)^2 + (2-1)^2 + (1-0.66)^2]$

$= 0.4356 + 1 + 0.1156 + 1 + 0.1156 = 1.6668.$

$C_2 \Rightarrow e_2^2 \Rightarrow [(3-3)^2 + (3-3)^2] = 0.$

Total square error

$\Rightarrow e_1^2 + e_2^2 = 1.6668$

Distance Calculation,

$X_1 (1, 0)$

$d(x_1, M_1) = \sqrt{(1-1)^2 + (0-0.66)^2} = \sqrt{0 + 0.4356} = 0.66$

$d(x_2, M_2) = \sqrt{(1-1)^2 + (0-3)^2} = \sqrt{4+9} = \sqrt{13}$

$= 3.605$

$X_2\ (0,1)$

$$d(x_2, M_1) = \sqrt{(0-1)^2 + (1-0.66)^2} = \sqrt{1+0.11} = 1.057$$

$$d(x_2, M_2) = \sqrt{(0-2)^2 + (1-3)^2} = \sqrt{9+4} = \sqrt{13} = 2.61$$

$X_3\ (2,1)$

$$d(x_3, M_1) = \sqrt{(2-1)^2 + (1-0.66)^2} = 1.056$$

$$d(x_3, M_2) = \sqrt{(2-3)^2 + (1-2)^2} = \sqrt{5} = 2.236$$

$X_4\ (3,3)$

$$d(x_4, M_1) = \sqrt{(3-1)^2 + (3-0.66)^2} = 9.4756.$$

$$d(x_4, M_2) = \sqrt{(3-3)^2 + (3-2)^2} = 0.$$

	M_1	M_2
X_1	0.66	3.605
X_2	1.056	2.605
X_3	1.056	2.236
X_4	9.4756	0
	C_1	C_2

$$\Rightarrow \quad C_1 = (x_1, x_3, x_2)$$

$$C_2 = (x_4).$$

The iteration stops here, as we get the same set of clusters as the previous step.

Hierarchical Clustering

- Cluster based on similarities/distances
- Distance measure between instances xr and xs

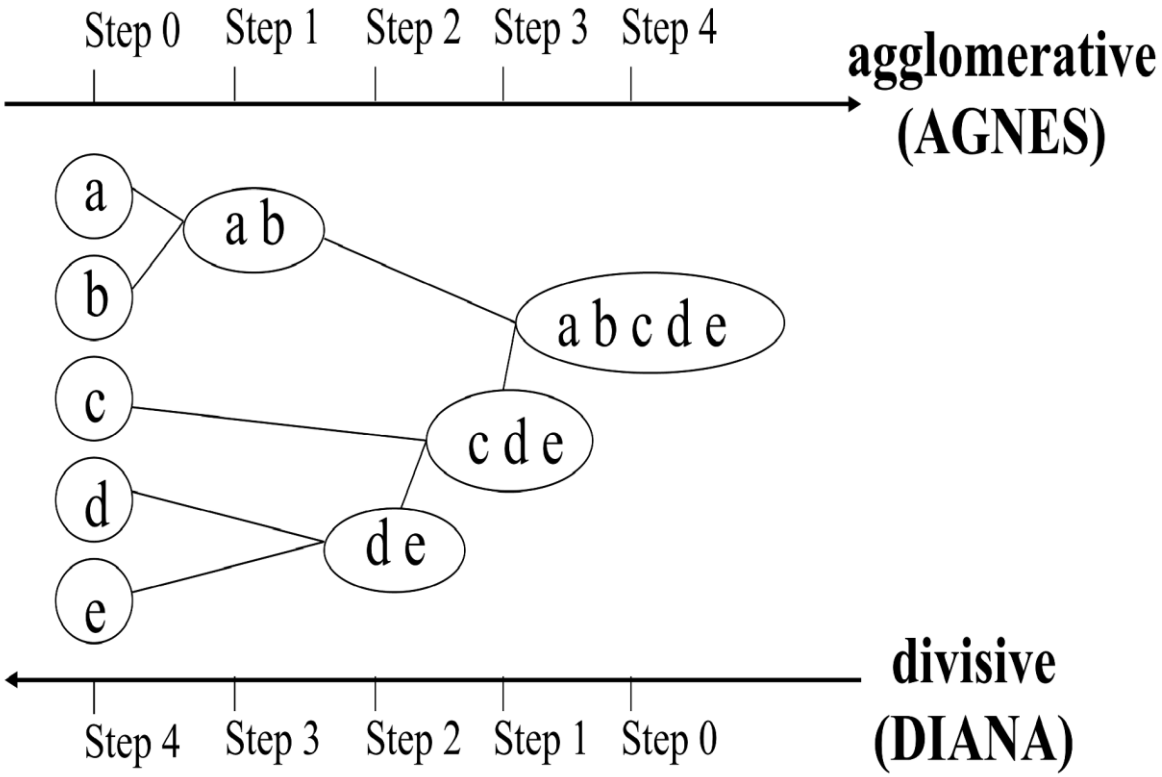
Minkowski distance (Lp) (Euclidean for p = 2)

$$d_m(\mathbf{x}^r, \mathbf{x}^s) = \left[\sum_{j=1}^{d} \left(x_j^r - x_j^s\right)^p\right]^{1/p}$$

City-block distance

$$d_{cb}(\mathbf{x}^r, \mathbf{x}^s) = \sum_{j=1}^{d} \left|x_j^r - x_j^s\right|$$

- Two important paradigms:
  - Bottom-up agglomerative clustering(Agglomerative Nesting-AGNES)
  - Top-down divisive clustering(Divisive Analysis-DIANA)



AgglomerativeClustering:

- Start with N groups each with one instance and merge two closest groups at eachiteration
- Distance between two groups Gi and Gj:
  - Single-link clustering:

$$d\left(G_i, G_j\right) = \min_{\mathbf{x}^r \in G_i, \mathbf{x}^s \in G_j} d\left(\mathbf{x}^r, \mathbf{x}^s\right)$$

  - Complete-link clustering:

$$d\left(G_i, G_j\right) = \max_{\mathbf{x}^r \in G_i, \mathbf{x}^s \in G_j} d\left(\mathbf{x}^r, \mathbf{x}^s\right)$$

  - Average-link clustering, centroid
  - **Dendrogram**: Decompose data objects into a several levels of nested partitioning(tree of clusters), called a dendrogram
  - A clustering of the data objects is obtained by cutting the dendrogram at thedesired level, then each connected component forms a cluster

Example of Single Linkage Clustering:

1. Given a datset, first find the distance matrix using Euclidean distance measure.
2. After finding the distance matrix, find the smallest element in the distance matix.
3. Merge these two points to form a cluster.
4. Next find the minium distance between the new cluster obtained with all other points.
5. Now frame the distance matrix and find the smallest value in the obtained distance matrix and then merge these points to form a cluster. The process repeats until all points are grouped into clusters.
6. Finally draw the dendogram.

Example:

	X	Y
P1	0.40	0.53
P2	0.22	0.38
P3	0.35	0.32
P4	0.26	0.19
P5	0.08	0.41
P6	0.45	0.30

Find clusters using single linkage technique.

Solution:

Step 1:-

Euclidean distance.

$$d(P_1,P_2) = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}.$$

$$= \sqrt{0.0549} = 0.234.$$

$d(P_1,P_3) = \sqrt{0.0466} = 0.215$

$d(P_2,P_3) = \sqrt{0.0205} = 0.1431.$

$d(P_1,P_4) = \sqrt{0.1352} = 0.3676$

$d(P_2,P_4) = \sqrt{0.0377} = 0.1941.$

$d(P_3,P_4) = \sqrt{0.025} = 0.1581.$

$d(P_1,P_5) = 0.34$

$d(P_2,P_5) = 0.14$

$d(P_3,P_5) = 0.28$

$d(P_4,P_5) = 0.27$

$P(P_1,P_6) = 0.23$

$d(P_2,P_6) = 0.25$

$d(P_3,P_6) = 011$

$d(P_4,P_6) = 0.22$

$d(P_5,P_6) = 0.37$

2) Get the distance matrix.

	P1	P2	P3	P4	P5	P6
P1	0					
P2	0.234	0				
P3	0.22	0.15	0			
P4	0.37	0.20	0.15	0		
P5	0.34	0.14	0.28	0.29	0	
P6	0.23	0.25	**0.11**	0.22	0.39	0

↳ smallest value

3) Find minimum element from the dist. matrix.

A) $P_3 P_6$:-

- Recalculate the dist matrix.
- To update the dist. matrix.,

$$\min(dist(P_3, P_6), P_1)$$
$$= \min(dist(P_3, P_1), (P_6, P_1))$$
$$= \min(0.22, 0.23) = 0.22$$

$$\min(dist(P_3, P_6), P_2)$$
$$= \min(dist(P_3, P_2), (P_6, P_2))$$
$$= \min(0.15, 0.25) = 0.15$$

$$\min(dist(P_3, P_6), P_4)$$
$$= \min(dist(P_3, P_4), (P_6, P_4))$$
$$= \min(0.15, 0.22) = 0.15$$

$$\min(dist(P_3, P_6), P_5)$$
$$= \min(dist(P_3, P_5), (P_6, P_5))$$
$$= \min(0.28, 0.39) = 0.28$$

Now, the new clusters formed are (P_3, P_6).

Updated values for clusters w.r.t $(P_3\ P_6)$ are :-

	P_1	P_2	P_3, P_6	P_4	P_5
P_1	0				
P_2	0.23	0			
P_3, P_6	0.22	0.15	0		
P_4	0.37	0.20	0.15	0	
P_5	0.34	**0.14** ↳ smallest element.	0.28	0.29	0

New cluster - $(P_2\ P_5)$:-

Update the dist. matrix.

$\min(dist(P_2, P_5), P_1)$

$= \min(dist[(P_2, P_1), (P_5, P_1)]$

$= \min(0.23, 0.34) = 0.23$

$\min(dist(P_2, P_5), (P_3, P_6))$

$= \min(dist[(P_2, (P_3 P_6)), (P_5, (P_3 P_6))]$

$= \min(0.15, 0.28) = 0.15$

$\min(dist(P_2, P_5), P_4)$

$= \min(dist[(P_2, P_4), (P_5, P_4)]$

$= \min(0.20, 0.29) = 0.20.$

Updated dist. matrix :-

	P_1	P_2P_5	P_3P_6	P_4
P_1	0			
P_2P_5	0.23	0		
P_3P_6	0.22	[0.15]	0	
P_4	0.37	0.20	0.18	0

New cluster - $[(P_3P_6)(P_2P_5)]$:-

Update the dist matrix

$\min\left(\text{dist}\,[(P_2P_5)(P_3P_6)], P_1\right)$

$= \min\left(\text{dist}\,[(P_2P_5), P_1][(P_3P_6), P_1]\right)$

$= \min(0.23, 0.22) = 0.22.$

$\min(\text{dist}\,[(P_2P_5)(P_3P_6)], P_4)$

$= \min(\text{dist}\,[(P_2P_5), P_4]$
$[(P_3P_6), P_4]$

$= \min(0.20, 0.18)$

$= 0.18$

Updated dist. matrix is:-

	P_1	P_2 P_5 P_3 P_6	P_4
P_1	0		
P_2 P_5 P_3 P_6	0.22.	0	
P_4	0.37	0.15	0

New cluster - $[P_4\ P_2\ P_5\ P_3\ P_6]$:-

Update the dist. matrix is-

$$\min(\text{dist}[P_2 P_5 P_3 P_6\ P_4], P_1)$$

$$= \min(\text{dist}[P_2 P_5 P_3 P_6, P_1], [P_4, P_1])$$

$$= \min(0.22, 0.37) = 0.22.$$

⇒ Updated dist. matrix:-

	P_1	$P_2\ P_5\ P_3\ P_6\ P_4$
P_1	0	
$P_2\ P_5\ P_3\ P_6\ P_4$	0.22.	0

Dendogram:-

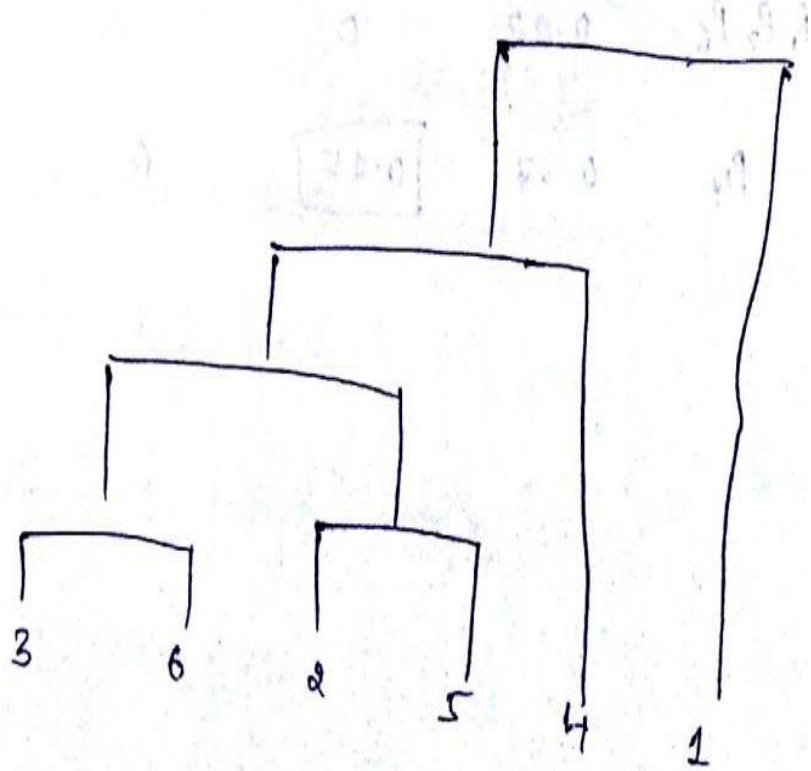
$P_3 P_6$

$P_2 P_5$

$(P_3 P_6)(P_2 P_5)$

$(P_3 P_6)(P_2 P_5) P_4$

$(P_3 P_6)(P_2 P_5) P_4 P_1$



Nonparametric Methods

- Parametric (single global model), semiparametric (small number of local models)
- Nonparametric: Similar inputs have similar outputs
- Functions (pdf, discriminant, regression) change smoothly
- Keep the training data;"let the data speak for itself"
- Given x, find a small number of closest training instances and interpolate from these
- Aka lazy/memory-based/case-based/instance-based learning

Density Estimation

- Given the training set X={xt}t drawn iid from p(x)
- Divide data into bins of size h
- Histogram:

$$\hat{p}(x) = \frac{\#\{x^t \text{ in the same bin as } x\}}{Nh}$$

- Naive estimator:

$$\hat{p}(x) = \frac{\#\{x - h < x^t \le x + h\}}{2Nh}$$

or

$$\hat{p}(x) = \frac{1}{Nh}\sum_{t=1}^{N} w\left(\frac{x - x^t}{h}\right) \quad w(u) = \begin{cases} 1/2 & \text{if } |u| < 1 \\ 0 & \text{otherwise} \end{cases}$$

with the weight function defined as,,

$$w(u) = \begin{cases} 1 & \text{if } |u| < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

This is as if each x^t has a symmetric region of influence of size h around it and contributes 1 for an x falling in its region. Then the nonparametric estimate is just the sum of influences of x^t whose regions include x. Because this region of influence is "hard" (0 or 1), the estimate is not a continuous function and has jumps at $x^t \pm h/2$.

Kernel Estimator:

- Kernel function, e.g., Gaussian kernel:

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{u^2}{2}\right]$$

- Kernel estimator (Parzen windows)

$$\hat{p}(x) = \frac{1}{Nh} \sum_{t=1}^{N} K\left(\left| \frac{x - x^t}{h} \right| \right)$$

The kernel function $K(\cdot)$ determines the shape of the influences and the window width h determines the width. Just like the naive estimate is the sum of "boxes," the kernel estimate is the sum of "bumps." All the x^t have an effect on the estimate at x, and this effect decreases smoothly as $|x - x^t|$ increases.

To simplify calculation, $K(\cdot)$ can be taken to be 0 if $|x - x^t| > 3h$. There exist other kernels easier to compute that can be used, as long as $K(u)$ is maximum for $u = 0$ and decreasing symmetrically as $|u|$ increases.

k-Nearest Neighbor Estimator:

- Instead of fixing bin width h and counting the number of instances, fix the instances(neighbors) k and check bin width,

$$\hat{p}(x) = \frac{k}{2Nd_k(x)}$$

$d_k(x)$, distance to kth closest instance to x.

The nearest neighbor class of estimators adapts the amount of smoothing to the local density of data. The degree of smoothing is controlled by k, the number of neighbors taken into account, which is much smaller than N, the sample size. Let us define a distance between a and b, for example, |a − b|, and for each x, we define,

$$d_1(x) \leq d_2(x) \leq \cdots \leq d_N(x)$$

to be the distances arranged in ascending order, from x to the points in the sample: $d_1(x)$ is the distance to the nearest sample, $d_2(x)$ is the distance to the next nearest, and so on. If x^t are the data points, then we define $d_1(x) = \min_t |x - x^t|$, and if i is the index of the closest sample, namely, $i = \arg\min_t |x - x^t|$, then $d_2(x) = \min_{j \neq i} |x - x^j|$, and so forth.

Generalization of multivariate data

- Kernel density estimator

$$\hat{p}(\mathbf{x}) = \frac{1}{Nh^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x}-\mathbf{x}^t}{h}\right)$$

- Multivariate Gaussian kernel
  - Spheric $$K(\mathbf{u}) = \left(\frac{1}{\sqrt{2\pi}}\right)^d \exp\left[-\frac{\|\mathbf{u}\|^2}{2}\right]$$
  - Ellipsoid $$K(\mathbf{u}) = \frac{1}{(2\pi)^{d/2}|\mathbf{S}|^{1/2}} \exp\left[-\frac{1}{2}\mathbf{u}^T\mathbf{S}^{-1}\mathbf{u}\right]$$

where S is the sample covariance matrix. This corresponds to using Mahalanobis distance instead of the Euclidean distance. It is also possible to have the distance metric local where S is calculated from instances in the vicinity of x, for example, some k closest instances.
Note that S calculated locally may be singular and PCA (or LDA, in thecase of classification) may be needed.

Hamming distance:

When the inputs are discrete, we can use Hamming distance, which counts the number of nonmatching attributes.

$$HD(\mathbf{x}, \mathbf{x}^t) = \sum_{j-1}^{d} 1(x_j \neq x_j^t)$$

where

$$1(x_j \neq x_j^t) = \begin{cases} 1 & \text{if } x_j \neq x_j^t \\ 0 & \text{otherwise} \end{cases}$$

$HD(\mathbf{x}, \mathbf{x}^t)$ is then used in place of $\|\mathbf{x} - \mathbf{x}^t\|$ or $(\mathbf{x} - \mathbf{x}^t)^T\mathbf{S}^{-1}(\mathbf{x} - \mathbf{x}^t)$ for kernel estimation or for finding the k closest neighbors.

Nonparametric Classification

- Estimate $p(\mathbf{x}|C_i)$ and use Bayes' rule
- Kernel estimator

$$\hat{p}(\mathbf{x} \mid C_i) = \frac{1}{N_i h^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x}-\mathbf{x}^t}{h}\right) r_i^t \qquad \hat{P}(C_i) = \frac{N_i}{N}$$

$$\hat{g}_i(\mathbf{x}) = \hat{p}(\mathbf{x} \mid C_i)\hat{P}(C_i) = \frac{1}{Nh^d} \sum_{t=1}^{N} K\left(\frac{\mathbf{x}-\mathbf{x}^t}{h}\right) r_i^t$$

- k-NN estimator

$$\hat{p}(\mathbf{x} \mid C_i) = \frac{k_i}{N_i V^k(\mathbf{x})} \qquad \hat{P}(C_i \mid \mathbf{x}) = \frac{\hat{p}(\mathbf{x} \mid C_i)\hat{P}(C_i)}{\hat{p}(\mathbf{x})} = \frac{k_i}{k}$$

The k-nn classifier assigns the input to the class having most examples among the k neighbors of the input. All neighbors have equal vote, and the class having the maximum number of voters among the k neighbors is chosen. Ties are broken arbitrarily or a weighted vote is taken. K is generally taken to be an odd number to minimize ties: confusion is generally between two neighboring classes. Again, the use of Euclidean distance corresponds to assuming uncorrelated inputs with equal variances, and when this is not the case a suitable discriminant metric should be used. One example is discriminant adaptive nearest neighbor where the optimal distance to separate classes is estimated locally.

Nonparametric Regression

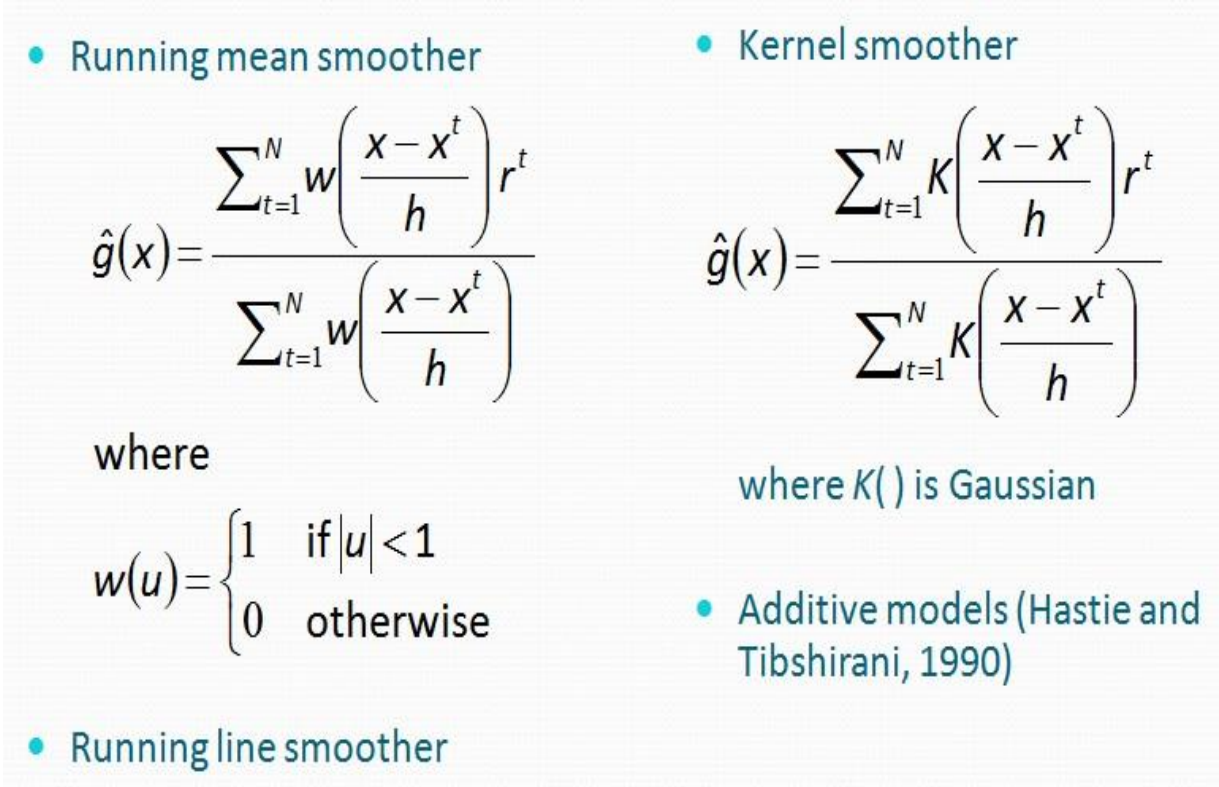
- Aka smoothing models
- Regressogram

$$\hat{g}(x) = \frac{\sum_{t=1}^{N} b(x, x^t) r^t}{\sum_{t=1}^{N} b(x, x^t)}$$

where

$$b(x, x^t) = \begin{cases} 1 & \text{if } x^t \text{ is in the same bin with } x \\ 0 & \text{otherwise} \end{cases}$$

Running Mean/Kernel Smoother:

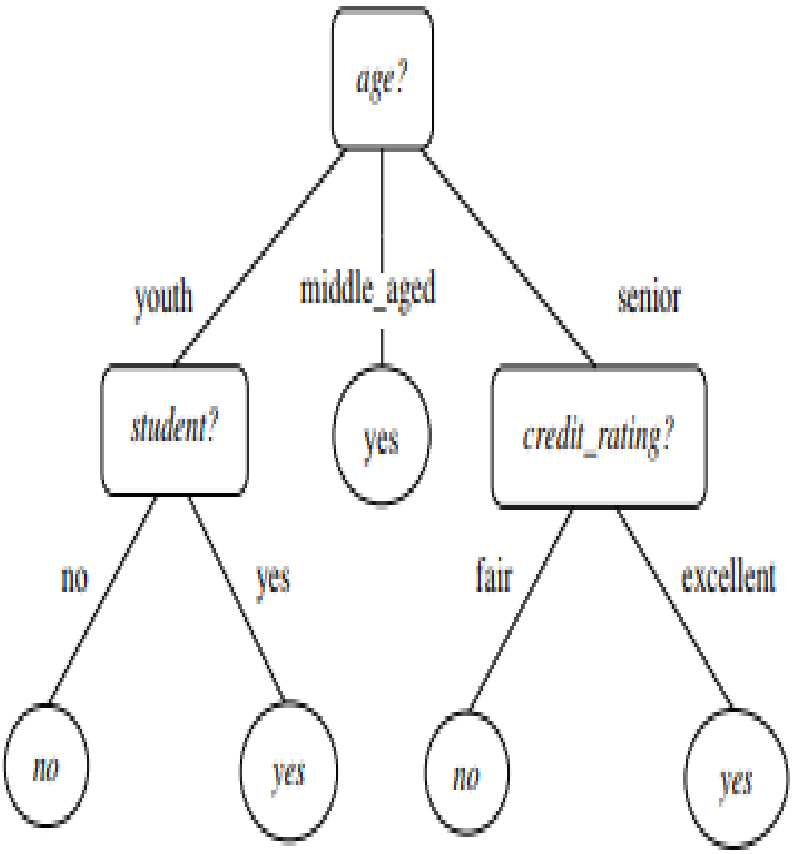


Instead of taking an average and giving a constant fit at a point, we can take into account onemore term in the Taylor expansion and calculate a linear fit. In the running line smoother, we can use the data points in the neighborhood, as defined by h or k, and fit a local regression line. In the locally weighted running line smoother, known as loess, instead of a hard definition of neighborhoods, we use kernel weighting such that distant points have less effect on error.

Decision Trees

Decision tree induction is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flowchart-like tree structure, where each **internal node** (non-leaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node. A typical decision tree is shown in Figure 8.2. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is

likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), whereas others can produce nonbinary trees.



A decision tree for the concept *buys_computer*, indicating whether an *AllElectronics* customer is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

Algorithm:

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition, *D*.

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

(1) create a node *N*;
(2) **if** tuples in *D* are all of the same class, *C*, **then**
(3) return *N* as a leaf node labeled with the class *C*;
(4) **if** *attribute_list* is empty **then**
(5) return *N* as a leaf node labeled with the majority class in *D*; // majority voting
(6) apply **Attribute_selection_method**(*D*, *attribute_list*) to **find** the "best" *splitting_criterion*;
(7) label node *N* with *splitting_criterion*;
(8) **if** *splitting_attribute* is discrete-valued **and**
multiway splits allowed **then** // not restricted to binary trees
(9) *attribute_list* ← *attribute_list* − *splitting_attribute*; // remove *splitting_attribute*
(10) **for each** outcome *j* of *splitting_criterion*
// partition the tuples and grow subtrees for each partition
(11) let D_j be the set of data tuples in *D* satisfying outcome *j*; // a partition
(12) **if** D_j is empty **then**
(13) attach a leaf labeled with the majority class in *D* to node *N*;
(14) **else** attach the node returned by **Generate_decision_tree**(D_j, *attribute_list*) to node *N*;
endfor
(15) return *N*;

Attribute Selection Measure:

Select the attribute with the highest information gain

Let p_i be the probability that an arbitrary tuple in D belongs to class C_i, estimated by $|C_{i,D}|/|D|$

Expected information (entropy) needed to classify a tuple in D:

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

Information needed (after using A to split D into v partitions) to classify D:

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

Information gained by branching on attribute A

$$Gain(A) = Info(D) - Info_A(D)$$

Example:

Class-Labeled Training Tuples from the *AllElectronics* Customer Database

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Induction of a decision tree using information gain. Table 8.1 presents a training set, D, of class-labeled tuples randomly selected from the *AllElectronics* customer database. (The data are adapted from Quinlan [Qui86]. In this example, each attribute is discrete-valued. Continuous-valued attributes have been generalized.) The class label attribute, *buys_computer*, has two distinct values (namely, {*yes*, *no*}); therefore, there are two distinct classes (i.e., $m = 2$). Let class C_1 correspond to *yes* and class C_2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node N is created for the tuples in D. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We compute the expected information needed to classify a tuple in D:

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940$$

Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* category "youth," there are two *yes* tuples and three *no* tuples. For the category "middle_aged," there are four *yes* tuples and zero *no* tuples. For the category "senior," there are three *yes* tuples and two *no* tuples. Using Eq. (8.2), the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

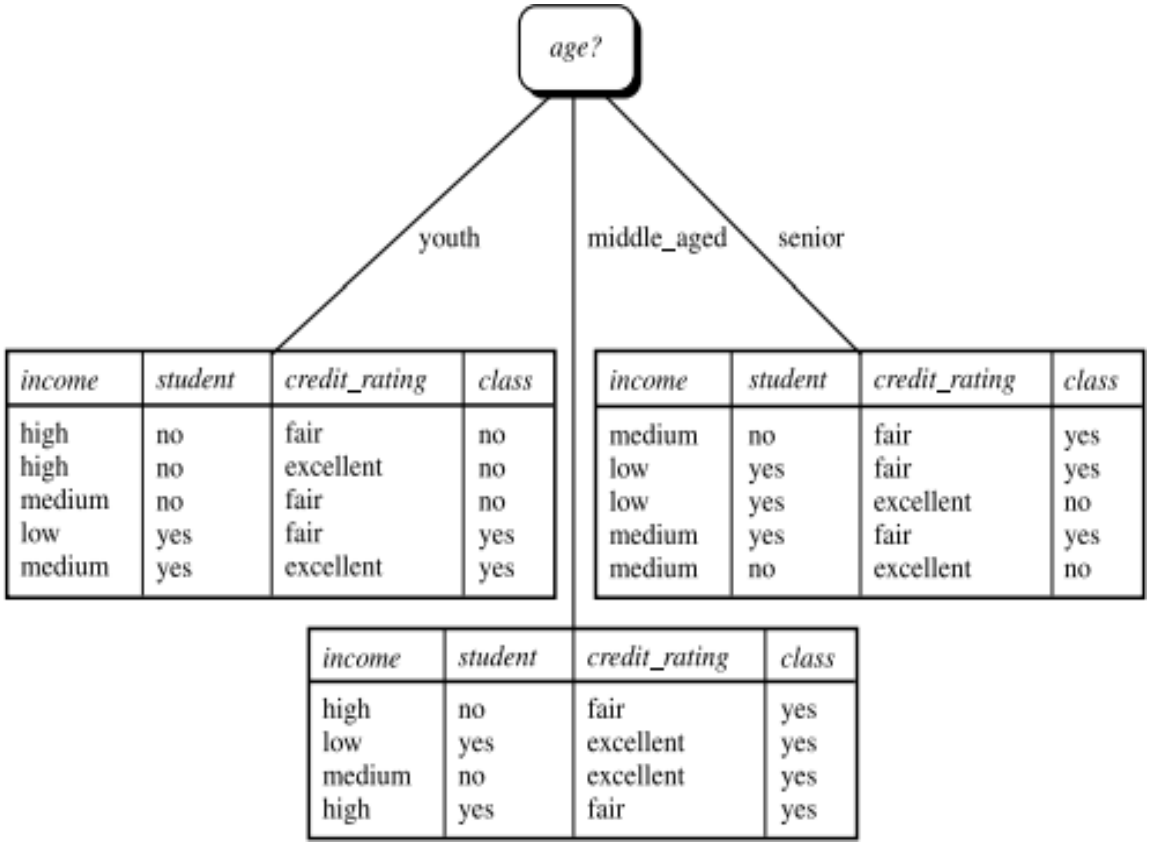
$$\begin{aligned} Info_{age}(D) = {} & \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right) \\ & + \frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4}\right) \\ & + \frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right) \\ & = 0.694 \end{aligned}$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246$$

Similarly, we can compute $Gain(income) = 0.029$
$Gain(student) = 0.151$
and $Gain(credit\_rating) = 0.048$

Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node N is labeled with *age*, and branches are grown for each of the attribute's values. The tuples are then partitioned accordingly, as shown in Figure 8.5. Notice that the tuples falling into the partition for *age* = *middle_aged* all belong to the same class. Because they all belong to class "*yes*," a leaf should therefore be created at the end of this branch and labeled "*yes*."

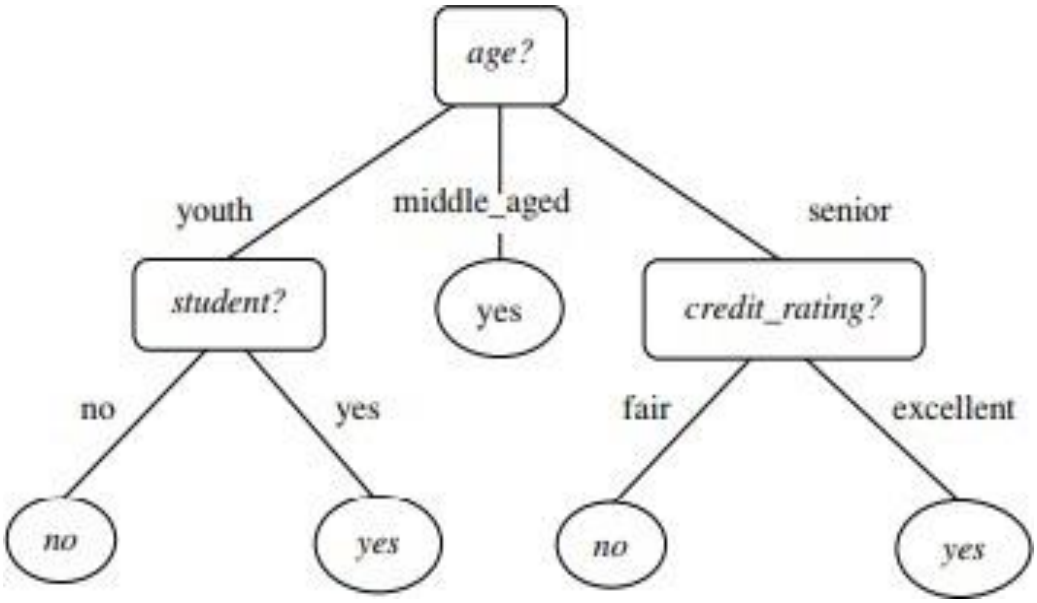


income	*student*	*credit_rating*	*class*
high	no	fair	no
high	no	excellent	no
medium	no	fair	no
low	yes	fair	yes
medium	yes	excellent	yes

income	*student*	*credit_rating*	*class*
medium	no	fair	yes
low	yes	fair	yes
low	yes	excellent	no
medium	yes	fair	yes
medium	no	excellent	no

income	*student*	*credit_rating*	*class*
high	no	fair	yes
low	yes	excellent	yes
medium	no	excellent	yes
high	yes	fair	yes

After further iterations, the final decision tree would be,



Univariate Trees

In a *univariate tree*, in each internal node, the test uses only one of the input dimensions. If the used input dimension, x_j, is discrete, taking one of n possible values, the decision node checks the value of x_j and takes the corresponding branch, implementing an n-way split. For example, if an attribute is color $\in$ {red, blue, green}, then a node on that attribute has three branches, each one corresponding to one of the three possible values of the attribute.

A decision node has discrete branches and a numeric input should be discretized. If x_j is numeric (ordered), the test is a comparison

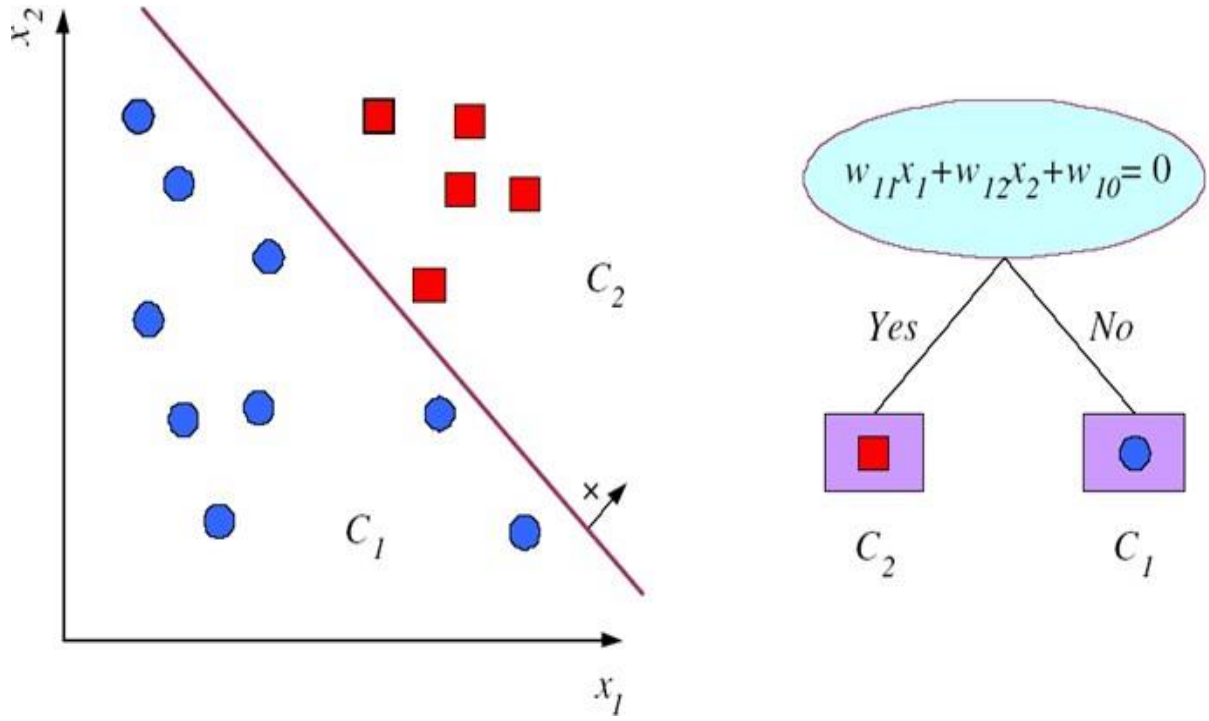
$$f_m(\mathbf{x}) : x_j > w_{m0}$$

where w_{m0} is a suitably chosen threshold value. The decision node divides the input space into two: $L_m = \{\mathbf{x}|x_j > w_{m0}\}$ and $R_m = \{\mathbf{x}|x_j \leq w_{m0}\}$; this is called a *binary split*. Successive decision nodes on a path from the root to a leaf further divide these into two using other attributes and generating splits orthogonal to each other. The leaf nodes define hyperrectangles in the input space (see figure 9.1).

Tree induction is the construction of the tree given a training sample. For a given training set, there exists many trees that code it with no error, and, for simplicity, we are interested in finding the smallest among them, where tree size is measured as the number of nodes in the tree and the complexity of the decision nodes. Finding the smallest tree is NP-complete (Quinlan 1986), and we are forced to use local search procedures based on heuristics that give reasonable trees in reasonable time.

Tree learning algorithms are greedy and, at each step, starting at the root with the complete training data, we look for the best split. This splits the training data into two or n, depending on whether the chosen attribute is numeric or discrete. We then continue splitting recursively with the corresponding subset until we do not need to split anymore, at which point a leaf node is created and labeled.

Multivariate Trees



Learning rules from data

From the decision tree, the following rules are identified.

If age=youth and student=no then buys_computer=no. If age=youth and student=yes then buys_computer=yes.If age=middle_aged then buys_computer=yes.
If age=senior and credit_rating=fair then buys_computer=no.
If age=senior and credit_rating=excellent then buys_computer=yes.

- Rule induction is similar to tree induction but
  - tree induction is breadth-first,
  - rule induction is depth-first; one rule at a time
- Rule set contains rules; rules are conjunctions of terms
- Rule covers an example if all terms of the rule evaluate to true for the example
- Sequential covering: Generate rules one at a time until all positive examples are covered.

```
Ripper(Pos,Neg,k)
  RuleSet ← LearnRuleSet(Pos,Neg)
  For k times
    RuleSet ← OptimizeRuleSet(RuleSet,Pos,Neg)
LearnRuleSet(Pos,Neg)
  RuleSet ← ∅
  DL ← DescLen(RuleSet,Pos,Neg)
  Repeat
    Rule ← LearnRule(Pos,Neg)
    Add Rule to RuleSet
    DL' ← DescLen(RuleSet,Pos,Neg)
    If DL'>DL+64
      PruneRuleSet(RuleSet,Pos,Neg)
      Return RuleSet
    If DL'<DL DL ← DL'
      Delete instances covered from Pos and Neg
  Until Pos = ∅
  Return RuleSet
```

```
PruneRuleSet(RuleSet,Pos,Neg)
  For each Rule ∈ RuleSet in reverse order
    DL ← DescLen(RuleSet,Pos,Neg)
    DL' ← DescLen(RuleSet-Rule,Pos,Neg)
    IF DL'<DL Delete Rule from RuleSet
  Return RuleSet
OptimizeRuleSet(RuleSet,Pos,Neg)
  For each Rule ∈ RuleSet
      DL0 ← DescLen(RuleSet,Pos,Neg)
      DL1 ← DescLen(RuleSet-Rule+
       ReplaceRule(RuleSet,Pos,Neg),Pos,Neg)
      DL2 ← DescLen(RuleSet-Rule+
       ReviseRule(RuleSet,Rule,Pos,Neg),Pos,Neg)
      If DL1=min(DL0,DL1,DL2)
       Delete Rule from RuleSet and
          add ReplaceRule(RuleSet,Pos,Neg)
      Else If DL2=min(DL0,DL1,DL2)
       Delete Rule from RuleSet and
          add ReviseRule(RuleSet,Rule,Pos,Neg)
  Return RuleSet
```

Linear Discrimination

Likelihood- vs. Discriminant-based Classification:

- Likelihood-based: Assume a model for $p(\mathbf{x}|C_i)$, use Bayes' rule to calculate

  $P(C_i|\mathbf{x})g_i(\mathbf{x}) = \log P(C_i|\mathbf{x})$

- Discriminant-based: Assume a model for $g_i(\mathbf{x}|\Phi_i)$; no density estimation
- Estimating the boundaries is enough; no need to accurately estimate the densities insidethe boundaries

Linear Discriminant:

- Linear discriminant:

$$g_i\left(\mathbf{x}\mid \mathbf{w}_i, w_{i0}\right)=\mathbf{w}_i^T\mathbf{x}+w_{i0}=\sum_{j=1}^{d} w_{ij}x_j+w_{i0}$$

- Advantages:
  - Simple: O(d) space/computation
  - Knowledge extraction: Weighted sum of attributes; positive/negative weights,magnitudes (credit scoring)
  - Optimal when $p(\mathbf{x}|C_i)$ are Gaussian with shared cov matrix; useful when classesare (almost) linearly separable

Generalized Linear Model:

- Quadratic discriminant:

$$g_i\left(\mathbf{x}\mid \mathbf{W}_i, \mathbf{w}_i, w_{i0}\right)=\mathbf{x}^T\mathbf{W}_i\mathbf{x}+\mathbf{w}_i^T\mathbf{x}+w_{i0}$$

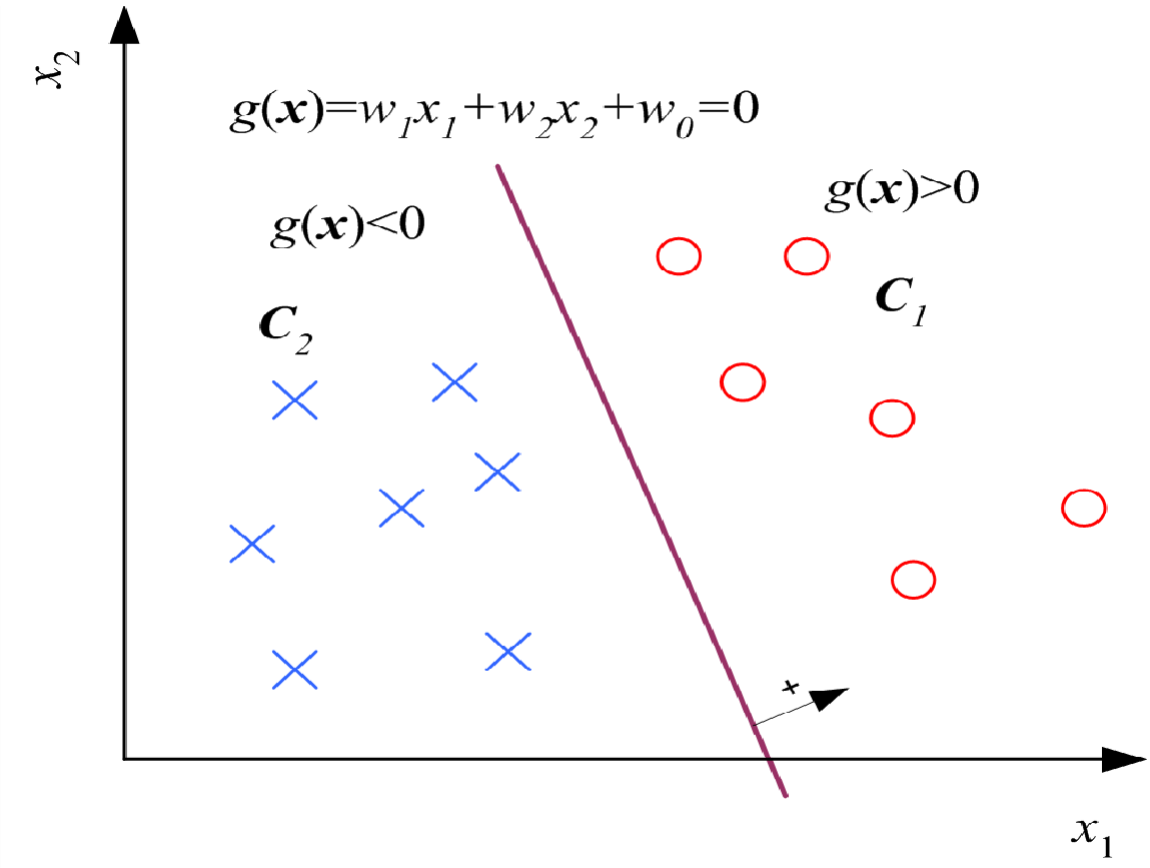
- Higher-order (product) terms:

$$z_1=x_1,\ z_2=x_2,\ z_3=x_1^2,\ z_4=x_2^2,\ z_5=x_1x_2$$

- Map from x to z using nonlinear basis functions and use a linear discriminant in z-space:

$$g_i(\mathbf{x}) = \sum_{j=1}^{k} w_{ij}\phi_j(\mathbf{x})$$

Two Classes:

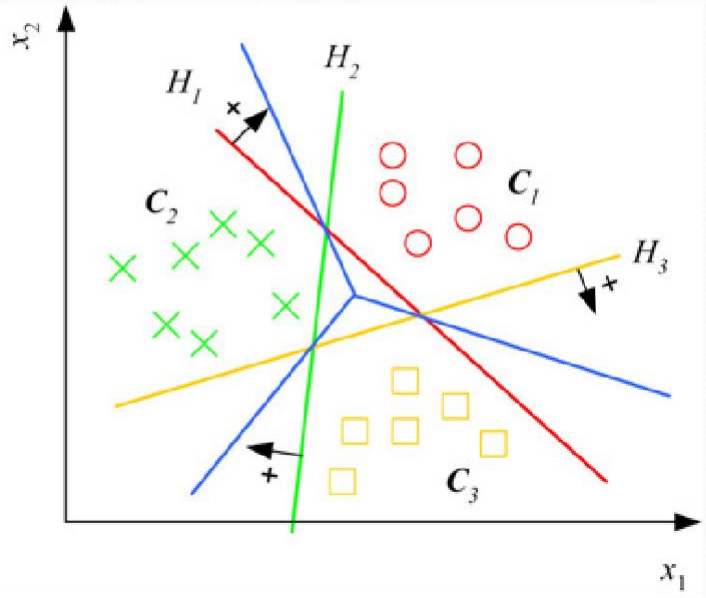


$$
\begin{aligned}
g(\mathbf{x}) &= g_1(\mathbf{x}) - g_2(\mathbf{x}) \\
&= \left(\mathbf{w}_1^T \mathbf{x} + w_{10}\right) - \left(\mathbf{w}_2^T \mathbf{x} + w_{20}\right) \\
&= \left(\mathbf{w}_1 - \mathbf{w}_2\right)^T \mathbf{x} + \left(w_{10} - w_{20}\right) \\
&= \mathbf{w}^T \mathbf{x} + w_0
\end{aligned}
$$

$$
\text{choose} \quad \begin{cases} C_1 & \text{if } g(\mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}
$$

Multiple Classes:

$$g_i(\mathbf{x} \mid \mathbf{w}_i, w_{i0}) = \mathbf{w}_i^T \mathbf{x} + w_{i0}$$

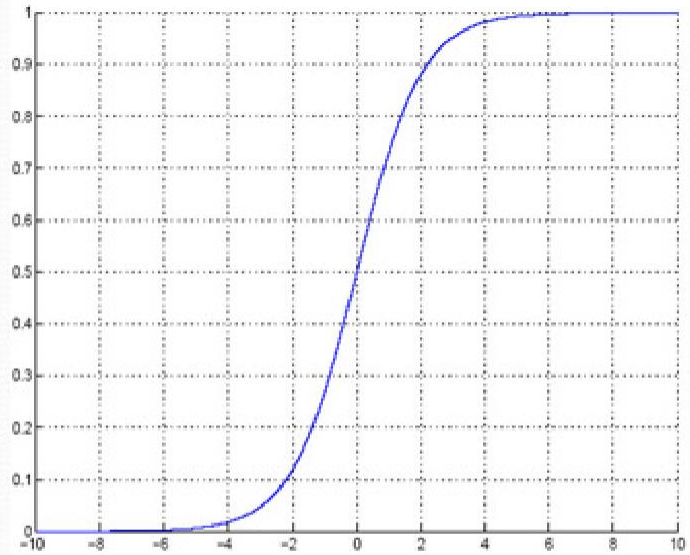


Choose C_i if

$$g_i(\mathbf{x}) = \max_{j=1}^{K} g_j(\mathbf{x})$$

Classes are linearly separable

Sigmoid (Logistic) Function:



1. Calculate $g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$ and choose C_1 if $g(\mathbf{x}) > 0$, or
2. Calculate $y = \text{sigmoid}(\mathbf{w}^T \mathbf{x} + w_0)$ and choose C_1 if $y > 0.5$

Gradient-Descent:

- E($\boldsymbol{w}$|X) is error with parameters $\boldsymbol{w}$ on sample X

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} E(\boldsymbol{w} \mid X)$$

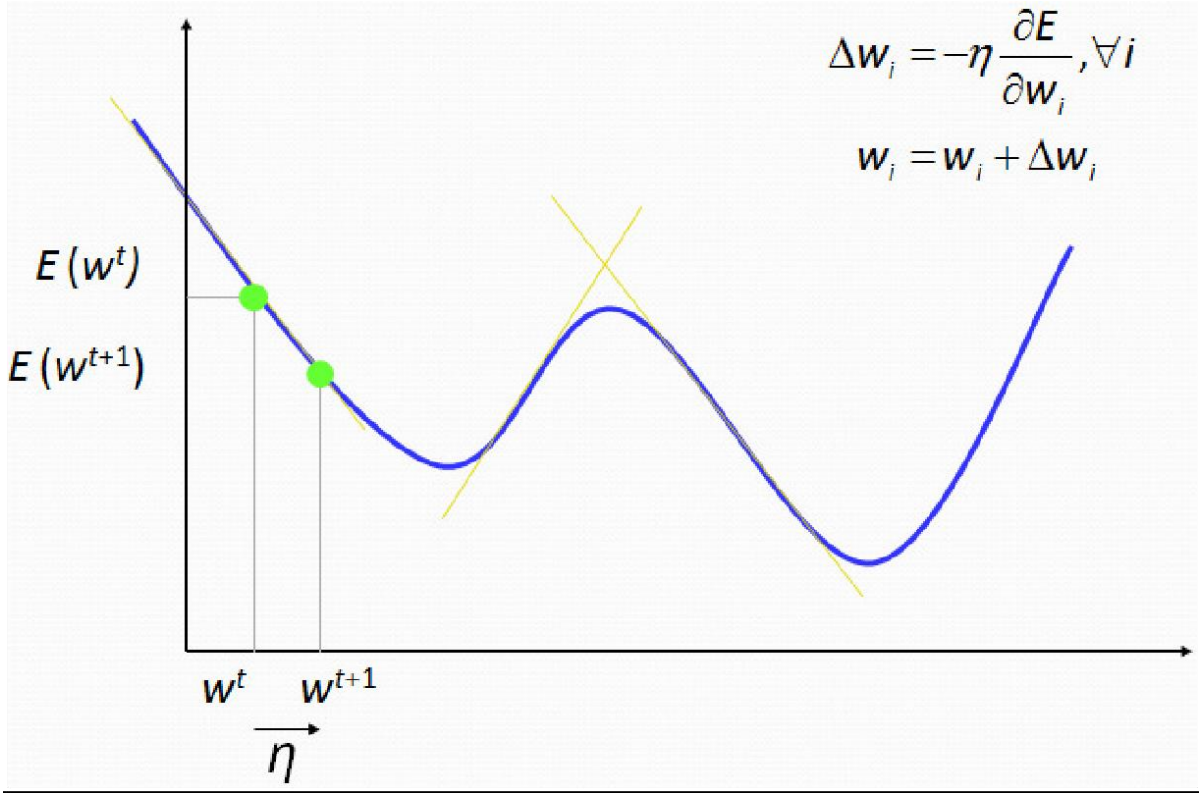
- Gradient

$$\nabla_w E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_d} \right]^T$$

- Gradient-descent:

  Starts from random $\boldsymbol{w}$ and updates $\boldsymbol{w}$ iteratively in the negative direction of gradient

---



Logistic Discrimination:

- Two classes: Assume log likelihood ratio is linear

$$\log\frac{p(\mathbf{x}\mid C_1)}{p(\mathbf{x}\mid C_2)} = \mathbf{w}^T\mathbf{x} + w_0^o$$

$$\text{logit}(P(C_1\mid\mathbf{x})) = \log\frac{P(C_1\mid\mathbf{x})}{1-P(C_1\mid\mathbf{x})} = \log\frac{p(\mathbf{x}\mid C_1)}{p(\mathbf{x}\mid C_2)} + \log\frac{P(C_1)}{P(C_2)}$$

$$= \mathbf{w}^T\mathbf{x} + w_0$$

$$\text{where } w_0 = w_0^o + \log\frac{P(C_1)}{P(C_2)}$$

$$y = \hat{P}(C_1\mid\mathbf{x}) = \frac{1}{1+\exp\left[-\left(\mathbf{w}^T\mathbf{x}+w_0\right)\right]}$$

UNIT – IV

Machine Learning – SCSA1601

UNIT IV MULTILAYER PERCEPTRONS

Structure of brain - Neural networks as a parallel processing - Perceptron - Multilayer perceptron - Backpropagation - Training procedures - Tuning the network size - Learning time

Structure of the Brain

The human brain is one of the most complicated things that we have studied in detail, and is, on the whole, poorly understood. We do not have satisfactory answers to the most fundamental of questions such as "what is my mind?" and "how do I think?". Nevertheless, we do have a basic understanding of the operation of the brain at a low level. It contains approximately ten thousand million (10^{11}) basic units, called neurons. Each of these neurons is connected to about ten thousand (10^4) others. To put this in perspective, imagine an Olympic-sized swimming pool, empty. The number of raindrops that it would take to fill the pool is approximately 10^{11}. You'd also need at least a dozen full address books if you were to be able to contact lo4 other people. The neuron is the basic unit of the brain, and is a stand-alone analogue logical processing unit. The neurons form two main types, local processing interneuron cells that have their input and output connections over about 100 microns, and output cells that connect different regions of the brain to each other, connect the brain to muscle, or connect from sensory organs into the brain. The operation of the neuron is a complicated and not fully understood process on a microscopic level, although the basic details are relatively clear. The neuron accepts many inputs, which are all added up in some fashion. If enough active inputs are received at once, then the neuron will be activated and "fire"; if not, then the neuron will remain in its inactive, quiet state. A representation of the basic features of a neuron is shown in the below figure.

The soma is the body of the neuron. Attached to the soma are long, irregularly shaped filaments, called dendrites. These nerve processes are often less than a micron in diameter, and have complex branching shapes. Their intricate shape resembles that of a tree in winter, without leaves, whose branches fork and fork again into finer structure. The dendrites act as the connections through which all the inputs to the neuron arrive. These cells are able to perform more complex functions than simple addition on the inputs they receive, but considering a simple summation is a reasonable approximation.

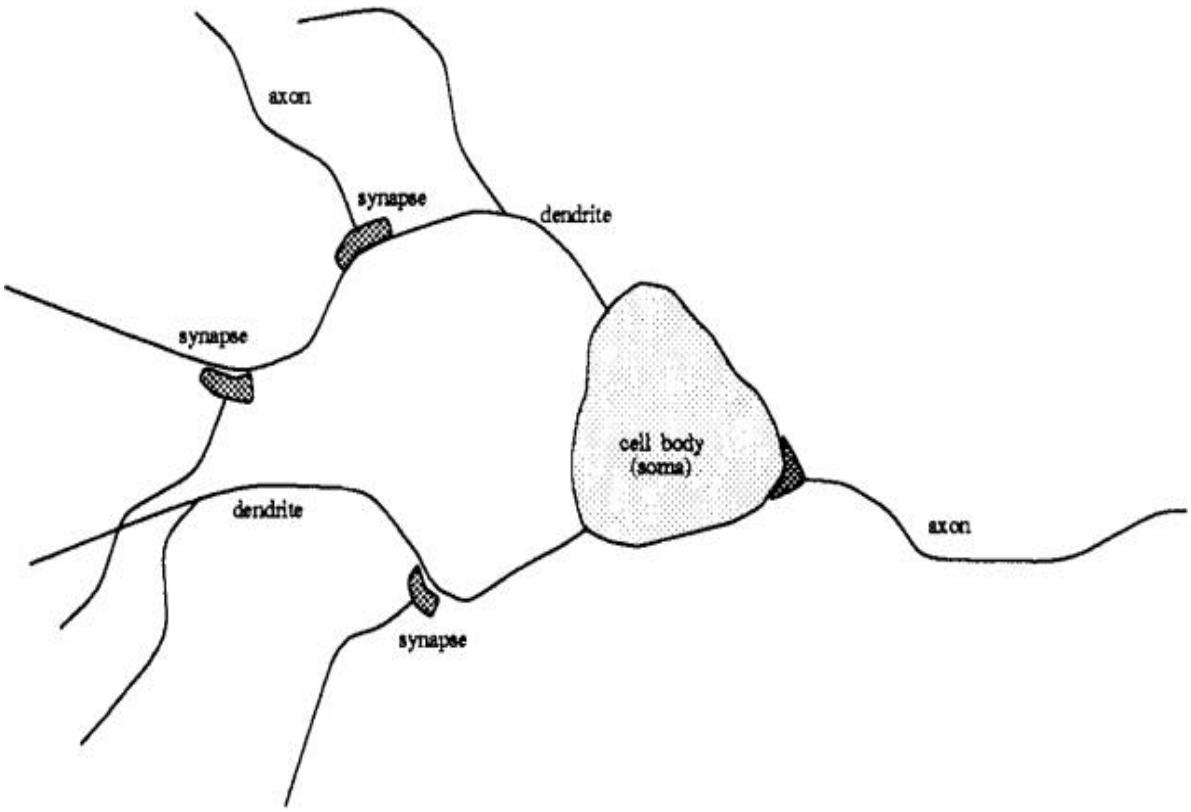


Fig. 4.1 The basic features of a biological neuron

Another type of nerve process attached to the soma is called an axon. This is electrically active, unlike the dendrite, and serves as the output channel of the neuron. Axons always appear on output cells, but are often absent from interneurons, which have both inputs and outputs on dendrites. The axon is a non-linear threshold device, producing a voltage pulse, called an action potential, that lasts about 1 millisecond (10^{-3}s) when the resting potential within the soma rises above a certain critical threshold. This action potential is in fact a series of rapid voltage spikes. The axon terminates in a specialised contact called a synapse that couples the axon with the dendrite of another cell. There is no direct linkage across the junction; rather, it is a temporary chemical one. The synapse releases chemicals called neurotransmitters when its potential is raised sufficiently by the action potential. It may take the arrival of more than one action potential before the synapse is triggered. The neurotransmitters that are released by the synapse diffuse across the gap, and chemically activate gates on the dendrites, which, when open, allow charged ions to flow. It is this flow of ions that alters the dendritic potential, and provides a voltage pulse on the dendrite, which is then conducted along into the next neuron body. Each dendrite may have many synapses acting on it, allowing massive interconnectivity to be achieved. At the synaptic junction, the number of gates opened on the dendrite depends on the number of neurotransmitters released. It also appears that some synapses excite the dendrite they affect, whilst others serve to inhibit it. This corresponds to altering the local potential of the

dendrite in a positive or negative direction. A single neuron will have many synaptic inputs on its dendrites, and may have many synaptic outputs connecting it to other cells.

Neural networks as a parallel processing

There are mainly two paradigms for *parallel processing*: In Single Instruction Multiple Data (SIMD) machines, all processors execute the same instruction but on different pieces of data. In Multiple Instruction Multiple Data (MIMD) machines, different processors may execute different instructions on different data. SIMD machines are easier to program because there is only one program to write. However, problems rarely have such a regular structure that they can be parallelized over a SIMD machine. MIMD machines are more general, but it is not an easy task to write separate programs for all the individual processors; additional problems are related to synchronization, data transfer between processors, and so forth. SIMD machines are also easier to build, and machines with more processors can be constructed if they are SIMD. In MIMD machines, processors are more complex, and a more complex communication network should be constructed for the processors to exchange data arbitrarily.

Assume now that we can have machines where processors are a little bit more complex than SIMD processors but not as complex as MIMD processors. Assume we have simple processors with a small amount of local memory where some parameters can be stored. Each processor implements a fixed function and executes the same instructions as SIMD processors; but by loading different values into the local memory, they can be doing different things and the whole operation can be distributed over such processors. We will then have what we can call Neural Instruction Multiple Data (NIMD) machines, where each processor corresponds to a neuron, local parameters correspond to its synaptic weights, and the whole structure is a neural network. If the function implemented in each processor is simple and if the local memory is small, then many such processors can be fit on a single chip.

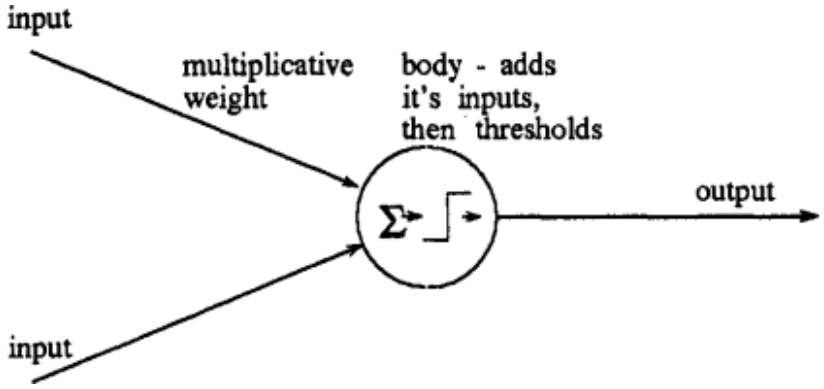
The problem now is to distribute a task over a network of such processors and to determine the local parameter values. This is where learning comes into play: We do not need to program such machines and determine the parameter values ourselves if such machines can learn from examples.

Thus, artificial neural networks are a way to make use of the parallel hardware we can build with current technology and—thanks to learning—they need not be programmed. Therefore, we also save ourselves the effort of programming them.

Perceptron

It performs a weighted sum of its inputs, compares this to some internal threshold level, and turns on only if this level is exceeded. If not, it stays off. Because the inputs are passed through the model neuron to produce the output, the system is known as a feedforward one. We need to formulate this mathematically. If there are n inputs, then there are n associated weights on the input lines. The model neuron calculates the weighted sum of its inputs; it takes the first input, multiplies it by the weight on that input line, then does the same for the next input, and so on, adding them all up at the end. This can be written as,

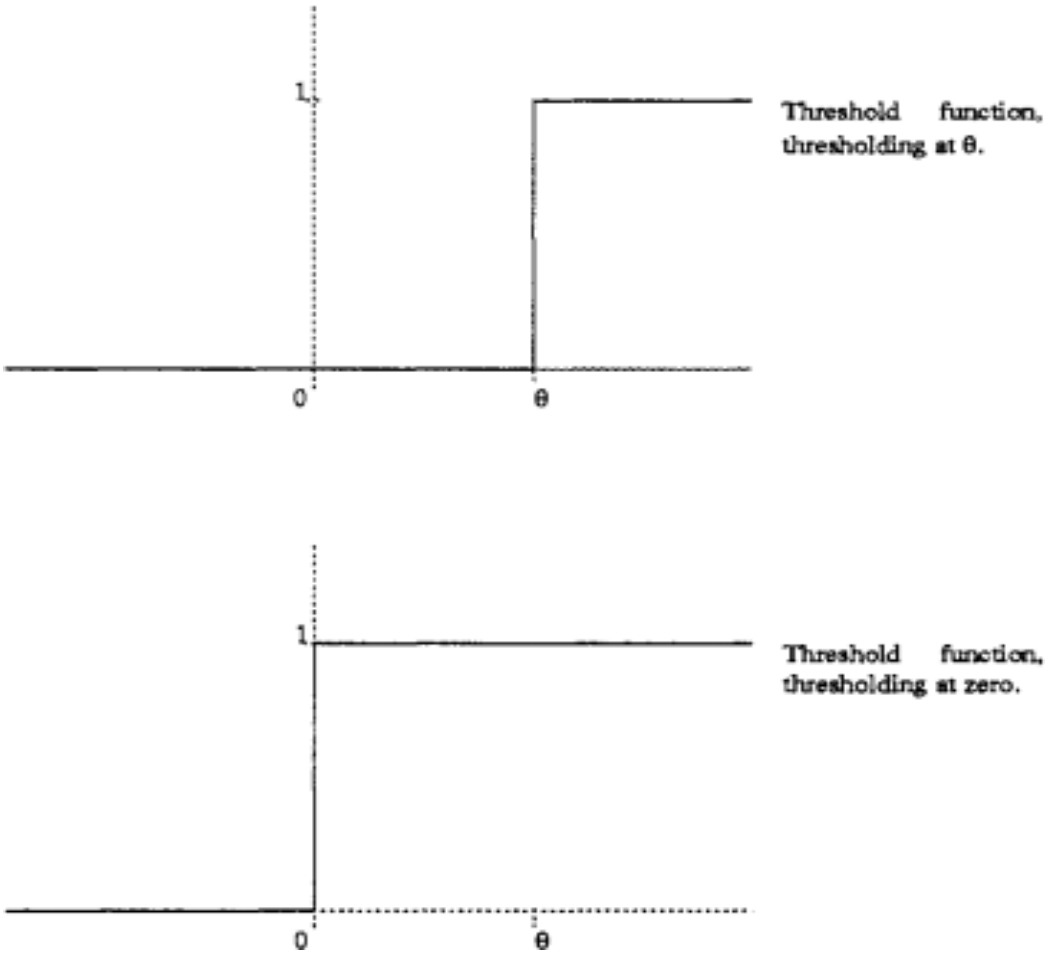
$$\begin{aligned}\text{total input} &= \text{weight on line 1} \times \text{input on 1} + \\ &\quad \text{weight on line 2} \times \text{input on 2} + \cdots + \\ &\quad \text{weight on line } n \times \text{input on } n \\ &= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + \cdots + w_nx_n \\ &= \sum_{i=1}^{n} w_ix_i\end{aligned}$$



Outline of the basic model.

This sum then has to be compared to a certain value in the neuron, the threshold value. This thresholding process is accomplished by comparison; if the sum is greater than the threshold value, then output a 1, if less, output a 0. This can be seen graphically in the figure given below, where the x-axis represents the input, and the y-axis the output.

The thresholding function is alternatively known as the ''step'' function, or the ''Heaviside'' function.



The thresholding function

Equivalently, the threshold value can be subtractec from the weighted sum, and the resulting value compared to zero; if the result is positive, then output a 1, else output a 0. Notice that the shape of the function is the same, but now the jump occurs at zero. The threshold effectivelyadds an offset to the weighted sum. An alternative way of achieving the same effect is to take the threshold out of the body of the model neuron and connect it to an extra input value that is fixedto be “on” all the time. In this case, rather than subtracting the threshold value from the weighted

sum, the extra input of +1 is multiplied by a weight equal to minus the threshold value, $-\theta$, and added in as well as all the other inputs-this is known as biasing the neuron. The value of -8 is therefore known as the neuron’s bias or ofSset. Both approaches are equivalent, and either is acceptable.

Calling the output y, we can write

$$y = f_h \left[\sum_{i=1}^{n} w_i x_i - \theta \right]$$

where f_h is a step function (actually known as the *Heaviside* function) and

$$f_h(x) = 1 \qquad x > 0$$

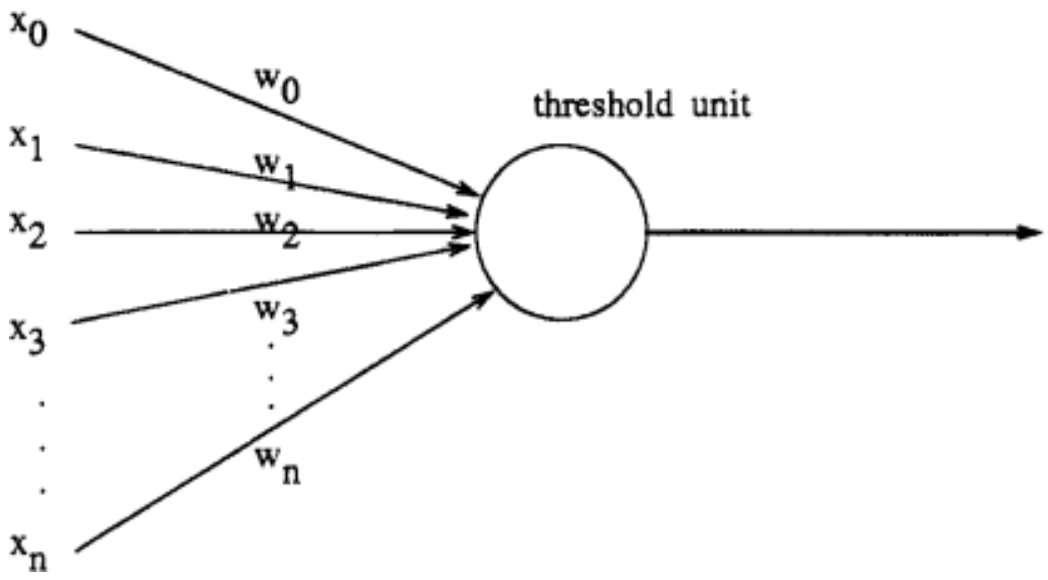
$$f_h(x) = 0 \qquad x \leq 0$$

so that it does what we want. Note that the function produces only a 1 or a 0, so that the neuron is either on or off.

If we use the approach of biasing the neuron, we can define an extra input, input 0, which is always set to be on, with a weight that represents the bias applied to the neuron. The equation describing the output can then be written as

$$y = f_h \left[\sum_{i=0}^{n} w_i x_i \right]$$

Notice that the lower limit of the summation has changed from 1 to 0, and that the value of the input xo is always set to 1. This model of the neuron, shown in the below figure, was proposedin 1943 by McCulloch and Pitts. Their model came about in much the same way as we have developed ours, and stemmed from their research into the behaviour of the neurons in the brain. It is important to look at the features of this McCulloch-Pitts neuron. It is a simple enough unit, thresholding a weighted sum of its inputs to get an output. It specifically does not take anyaccount of the complex patterns and timings of actual nervous activity in real neural systems, nor does it have any of the complicated features found in the body of biological neurons.



Details of basic Model

This ensures its status as a model, and not a copy, of a real neuron, and makes it possible to implement on a digital computer. This is the strength of the model-now we need to investigate what can be achieved using this simple design. The arrangement of the connections between the neurons is important, but, continuing our trend of choosing simple models to get an idea of what is happening in a complicated red-world situation, we shall for the time being consider only one layer of neurons, where we study the outputs of the neurons under a known set of inputs. The model neurons, connected up in a simple fashion, were given the name "perceptrons" by Frank Rosenblatt in 1962. He pioneered the simulation of neural networks on digital computers, as well as their formal analysis. In his book "Principles of Neurodynamics ", he describes these perceptrons as simplified networks in which certain properties of red nervous systems axe exaggerated whilst others are ignored. He stated that they are not intended to serve as detailed copies of any real nervous system; in other words, he realised at this early stage that he wasdealing with a basic model. This fact is often lost in the popular press as the idea of computer "brains", based on these techniques, grabs the imagination. We are not attempting to build computer brains, nor are we trying to mimic parts of red brains-rather we are aiming to discover the properties of models that take their behaviour from extremely simplified versions of natural neural systems, usually on a massively reduced scale as well. Whereas the brain has at least 10^{11} neurons, each connected to 10^4 others, we are concerned here with maybe a few hundred neuronsat most, connected to a few thousand input lines.

The learning paradigm can be summarised as follows:

- set the weights and thresholds randomly
- present an input

- calculate the actual out put by taking the thresholded value of the weighted sum of theinputs
- alter the weights to reinforce correct decisions and discourage incorrect decisions-i.e.reduce the error
- present the next input etc

The perceptron learning algorithm

Perceptron Learning Algorithm

---

1. Initialise weights and threshold

Define $w_i(t), (0 \leq i \leq n)$, to be the weight from input i at time t, and θ to be the threshold value in the output node. Set w_0 to be $-\theta$, the bias, and x_0 to be always 1.

Set $w_i(0)$ to small random values, thus initialising all the weights and the threshold.

2. Present input and desired output

Present input $x_0, x_1, x_2, \ldots, x_n$ and desired output $d(t)$

3. Calculate actual output

$$y(t) = f_h \left[\sum_{i=0}^{n} w_i(t) x_i(t) \right]$$

4. Adapt weights

$$\begin{array}{rll} \text{if correct} & w_i(t+1) = & w_i(t) \\ \text{if output 0, should be 1 (class A)} & w_i(t+1) = & w_i(t) + x_i(t) \\ \text{if output 1, should be 0 (class B)} & w_i(t+1) = & w_i(t) - x_i(t) \end{array}$$

Note that weights are unchanged if the net makes the correct decision. Also, weights are not adjusted on input lines which do not contribute to the incorrect response, since each weight is adjusted by the value of the input on that line, x_i, which would be zero.

This is the basic perceptron algorithm. However, various modifications have been suggested to this basic algorithm. The first is to introduce a multiplicative factor of less than one into the weight adaption term. This has the effect of slowing down the change in the weights, making the network take smaller steps towards the solution. This alteration to the algorithm entails replacing step 4 with the following:

4. Adapt weights—modified version

$$\begin{array}{rrcl} \textbf{if correct} & w_i(t+1) & = & w_i(t) \\ \textbf{if output 0, should be 1 (class A)} & w_i(t+1) & = & w_i(t) + \eta x_i(t) \\ \textbf{if output 1, should be 0 (class B)} & w_i(t+1) & = & w_i(t) - \eta x_i(t) \end{array}$$

where $0 \leq \eta \leq 1$, a positive gain term that controls the adaption rate.

Another algorithm of a similar nature was suggested by Widrow and Hoff. They realised that it would be best to change the weights by a lot when the weighted sum is a long way from the desired value, whilst altering them only slightly when the weighted sum is close to that required to give the correct solution. They proposed a learning rule known as the Widrow-Hoff delta rule, which calculates the difference between the weighted sum and the required output, and calls that the *error*. Weight adjustment is then carried out in proportion to that error. This means that during the learning process, the output from the unit is *not* passed through the step function—however, actual classification is effected by using the step function to produce the +1 or 0 indication as before.

The error term Δ can be written

$$\Delta = d(t) - y(t)$$

where $d(t)$ is the desired response of the system, and $y(t)$ is the actual response. This takes care of the addition or subtraction, since if the desired output is 1 and the actual output is 0, $\Delta = +1$ and

so the weights are increased. Conversely, if the desired output is 0 and the actual output is $+1$, $\Delta = -1$ and so the weights will be decreased. Note that weights are unchanged if the net makes the correct decision, since $d(t) - y(t) = 0$.

The learning algorithm is basically the same as for the basic perceptron, except this time step 4 is replaced by

4. Adapt weights—Widrow-Hoff delta rule

$$\begin{aligned} \Delta &= d(t) - y(t) \\ w_i(t+1) &= w_i(t) + \eta \Delta x_i(t) \\ d(t) &= \begin{cases} +1, & \text{if input from class A} \\ 0, & \text{if input from class B} \end{cases} \end{aligned}$$

where $0 \leq \eta \leq 1$, a positive gain function that controls the adaption rate

Neuron units using this learning algorithm were called ADALINEs (adaptive linear neurons) by Widrow, who also connected many of them together into a many-ADALINE structure, or MADALINE.

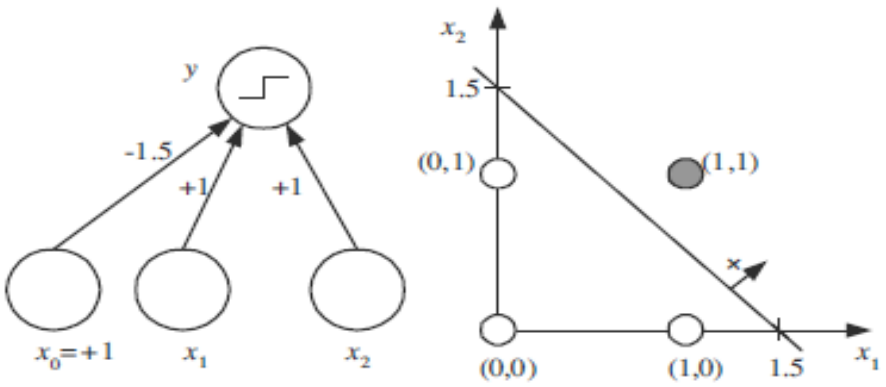
Another alternative proposed is to use inputs that are not 0 or 1 (binary), but are instead -1 or $+1$, known as *bipolar*. Using binary inputs means that input lines with 0's on them are not trained, whereas bipolar values allow all the inputs to be trained each time. This simple alteration helps to speed up the convergence process, but often leads to confusion in the literature as some authors discuss binary inputs and others bipolar ones. Effectively, they are equivalent, and the use of one or the other is usually a matter of personal preference.

Learning Boolean Functions:

In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise. Therefore, it can be seen as a two-class classification problem. As an example, for learning to AND two inputs, the table of inputs and required outputs is given in table . An example of a perceptron that implements AND and its

Table · Input and output for the AND function.

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1



The perceptron that implements AND and its geometric interpretation.

geometric interpretation in two dimensions is given in figure 11.4. The discriminant is

$$y = s(x_1 + x_2 - 1.5)$$

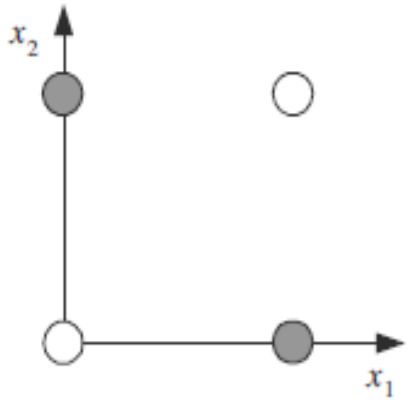
that is, $\boldsymbol{x} = [1, x_1, x_2]^T$ and $\boldsymbol{w} = [-1.5, 1, 1]^T$. Note that $y = s(x_1+x_2-1.5)$ satisfies the four constraints given by the definition of AND function in table 11.1, for example, for $x_1 = 1, x_2 = 0$, $y = s(-0.5) = 0$. Similarly it can be shown that $y = s(x_1 + x_2 - 0.5)$ implements OR.

Though Boolean functions like AND and OR are linearly separable and are solvable using the perceptron, certain functions like XOR are not.

The problem is not linearly separable. This can also be proved by noting that there are no w_0, w_1, and w_2 values that

Input and output for the XOR function.

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0



XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

satisfy the following set of inequalities:

$$
\begin{aligned}
& & w_0 & \leq 0 \\
& w_2+ & w_0 & > 0 \\
w_1+ & & w_0 & > 0 \\
w_1+ & w_2+ & w_0 & \leq 0
\end{aligned}
$$

This result should not be very surprising to us since the VC dimension of a line (in two dimensions) is three. With two binary inputs there are four cases, and thus we know that there exist problems with two inputs that are not solvable using a line; XOR is one of them.

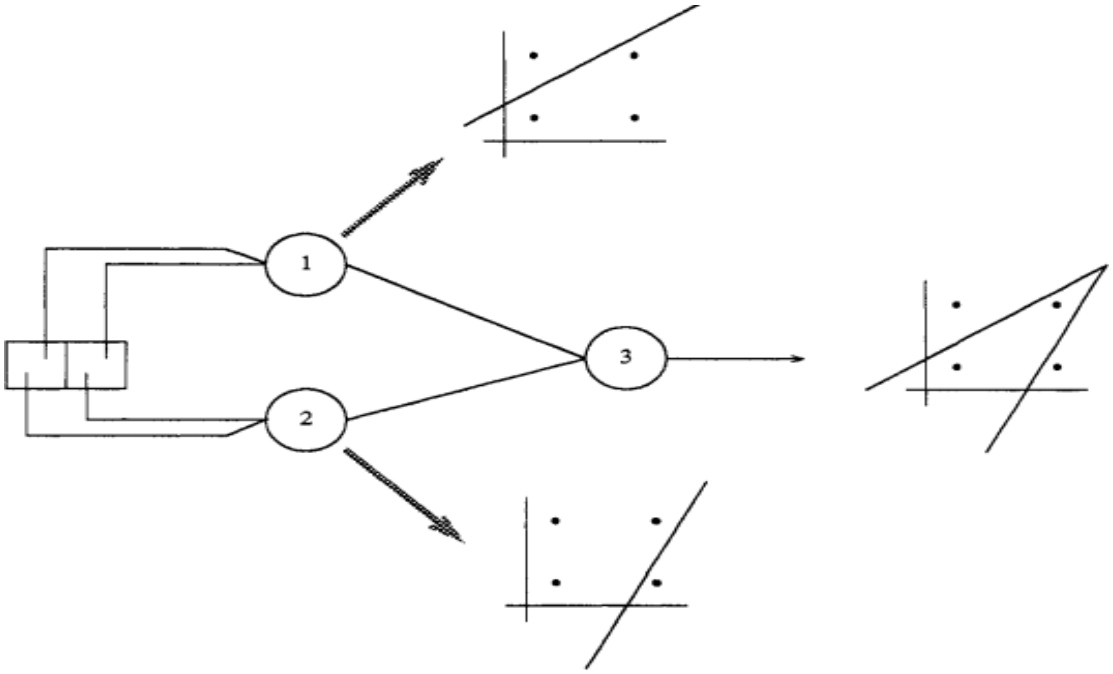
Multilayer perceptron

ALTERING THE PERCEPTRON MODEL

1 The Problem

How are we to overcome the problem of being unable to solve linearly inseparable problems with our perceptron? An initial approach would be to use more than one perceptron, each set up to identify small, linearly separable sections of the inputs, then combining their outputs into another perceptron, which would produce a final indication of the class to which the input belongs. This approach to the XOR problem is shown in figure 4.1.

This seems fine on first examination, but a moment's thought will show that this arrangement of perceptrons in layers will be unable to learn. Each neuron in the structure still takes the weighted sum of its inputs, thresholds it, and outputs either a one or a zero. For the perceptrons in the first layer, the inputs come from the actual inputs to the network, while the perceptrons in the second layer take as their inputs the outputs from the first layer. This means



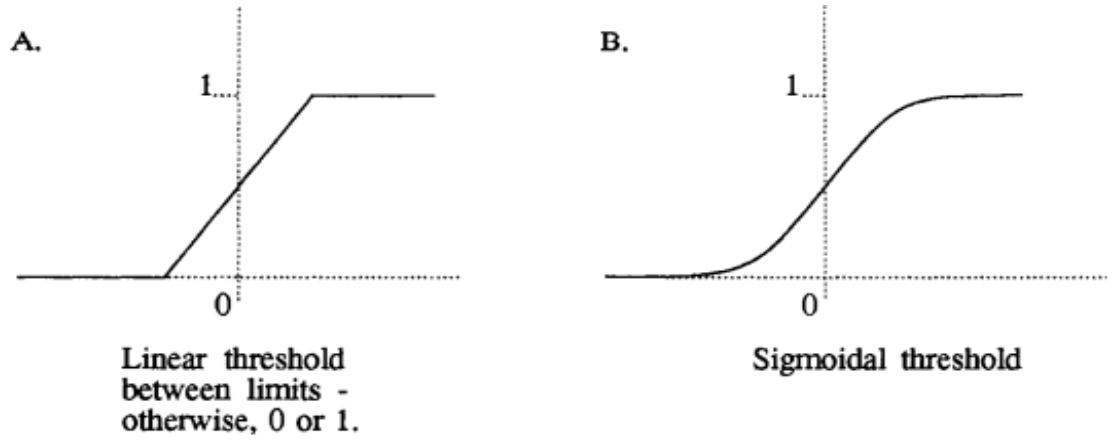
Combining perceptrons can solve the XOR problem: perceptron 1 detects when the pattern corresponding to (0,1) is present, and the other detects when (1,0) is there. Combined, these two facts allow perceptron 3 to classify the input correctly. They have to be set up correctly in the first place, however; they cannot learn to produce this classification.

that the perceptrons in the second layer do not know which of the real inputs were on or not; they are only aware of input from the first layer. Since learning corresponds to strengthening the connections between active inputs and active units (refer to section 3.3), it is impossible to strengthen the correct parts of the network, since the actual inputs are effectively masked off from the output units by the intermediate layer. The two-state neuron, being "on" or "off", gives us no indication of the scale by which we need to adjust the weights, and so we cannot make a reasonable adjustment. Weighted inputs that only just turn a neuron on should not be altered to the same extent as those in which the neuron is definitely turned on, but we have no way of finding out what the situation is. In other words, the hard-limiting threshold function (figure 3.3) removes the information that is needed if the network is to successfully learn. This difficulty is known as the *credit assignment* problem, since it means that the network is unable to determine which of the input weights should be increased and which should not, and so is unable to work out what changes should be made to produce a better solution next time.

The Solution

The way around the difficulty imposed by using the step function as the thresholding process is to adjust it slightly, and use a slightly different non-linearity. If we smooth it out, so that it more or less turns on or off, as before, but has a sloping region in the middle that will give us some information on the inputs, we will be able to determine when we need to strengthen or weaken the relevant weights. This means that the network will be able to learn, as required. A couple of possibilities for the new thresholding function are shown in figure 4.2.

In both cases, the value of the output will be practically one if the weighted sum exceeds the threshold by a lot, and conversely, it will be practically zero if the weighted sum is much less than the threshold value. However, in the case when the threshold and the weighted sum are almost the same, the output from the neuron will have a value somewhere between the two extremes. This means that



Two possible thresholding functions.

the output from the neuron is able to be related to its inputs in a more useful and informative way.

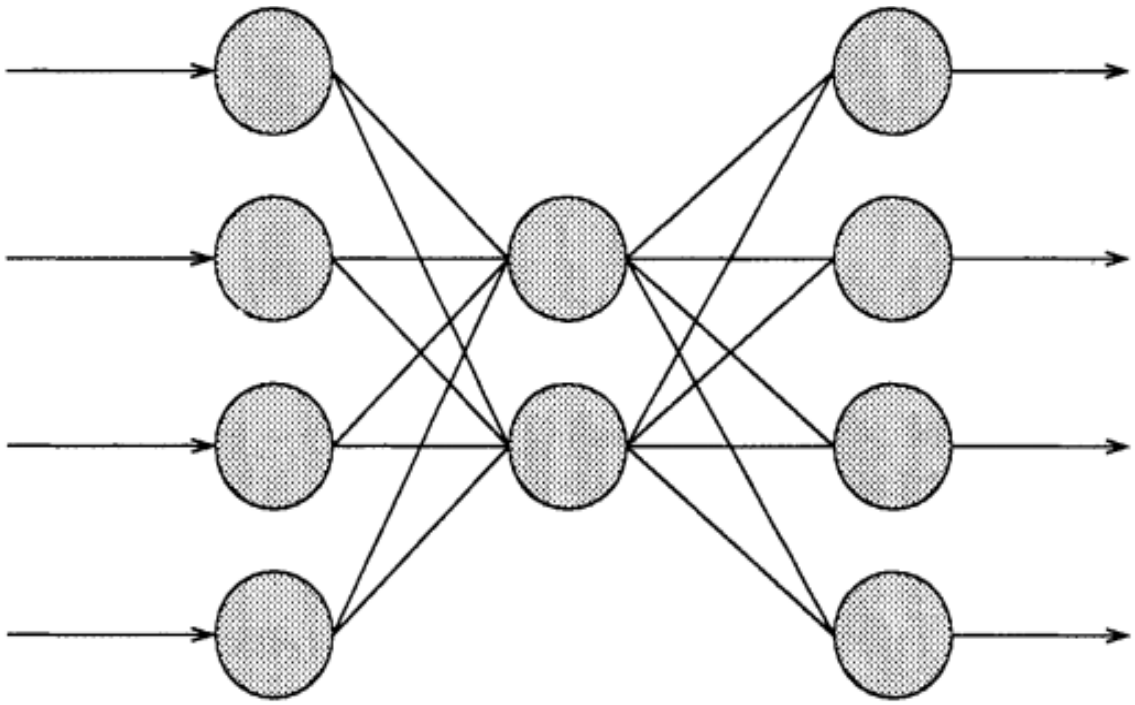
Notice that we have altered our model to try and overcome a particular difficulty by tracing the root of the problem, the hard-limiting thresholding that masks the inputs from the outputs, and then adjusting the model so that this can be solved. We have kept many of the essential features the same; each neuron still calculates the weighted sum, and thresholds it. However the input is now not

simply on or off, but lies within a range, although the thresholding function that we are using approximates to the step function in many ways, especially at the extremes of its range. The solution that we have adopted is therefore one tailored to our particular problem, and it would be foolish of us to say that *real* biological neurons also work in this way. We are looking at an interesting construction of model neurons, and *not* at a small version of a real brain. This may appear obvious to the reader, but it is surprising how many false claims are made about models that have their roots in biological systems, and a timely reminder can do no harm.

We have to use a non-linear thresholding function, since layers of perceptron units using linear functions are no more powerful than a suitably chosen single layer. This is because each layer would be performing a purely linear operation on its inputs, which could be condensed into one operation. This is easiest to see with a simple example. Changing scale is a linear operation, since all things are affected by an equal amount. If a network scaled the input by 5 times in the first layer, and by 2 times in the second, that is exactly equivalent to one layer scaling the whole thing by 10 times.

THE NEW MODEL

The adapted perceptron units are arranged in layers, and so the new model is naturally enough termed the *multilayer perceptron.* The basic details are shown in figure.



The multilayer perceptron: our new model.

Our new model has three layers; an input layer, an output layer, and a layer in between, not connected directly to the input or the output, and so called the *hidden* layer. Each unit in the hidden layer and the output layer is like a perceptron unit, except that the thresholding function is the one shown in figure , the sigmoid function B and not the step function as before. The units in the input layer serve to distribute the values they receive to the next layer, and so do not perform a weighted sum or threshold. Because we have modified the single-layer perceptron by changing the non-linearity from a step function to a sigmoid function, and added a hidden layer, we are forced to alter our learning rule as well.

THE MULTILAYER PERCEPTRON ALGORITHM

The algorithm for the multilayer perceptron that implements the back-propagation training rule is shown below. It requires the units to have thresholding non-linear functions that are continuously differentiable, i.e. smooth everywhere. We have assumed the use of the sigmoid function, $f(net) = 1/(1 + e^{-k\ net})$ since it has a simple derivative.

Multilayer Perceptron Learning Algorithm

---

1. Initialise weights and thresholds

Set all weights and thresholds to small random values.

2. Present input and desired output

Present input $X_p = x_0, x_1, x_2, \ldots, x_{n-1}$ and target output $T_p = t_0, t_1, \ldots, t_{m-1}$ where n is the number of input nodes and m is the number of output nodes. Set w_0 to be $-\theta$, the bias, and x_0 to be always 1. For pattern association, X_p and T_p represent the patterns to be associated. For classification, T_p is set to zero except for one element set to 1 that corresponds to the class that X_p is in.

3. Calculate actual output

Each layer calculates

$$y_{pj} = f\left[\sum_{i=0}^{n-1} w_i x_i\right]$$

and passes that as input to the next layer. The final layer outputs values o_{pj}.

4. Adapt weights

Start from the output layer, and work backwards.

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_{pj} o_{pj}$$

$w_{ij}(t)$ represents the weights from node i to node j at time t, η is a gain term, and δ_{pj} is an error term for pattern p on node j.

For output units

$$\delta_{pj} = k o_{pj}(1 - o_{pj})(t_{pj} - o_{pj})$$

For hidden units

$$\delta_{pj} = k o_{pj}(1 - o_{pj}) \sum_{k} \delta_{pk} w_{jk}$$

where the sum is over the k nodes in the layer above node j.

Backpropagation

Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops.

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Method:

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) **for** each training tuple $\boldsymbol{X}$ in D {
(4) // Propagate the inputs forward:
(5) **for** each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) **for** each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) **for** each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) **for** each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) **for** each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) **for** each bias θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }

Backpropogation Algorithm

Example:

The figure given below shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 1, along with the first training tuple, **X= (1, 0, 1),** with a class label of 1 (target value T_j). This example shows the calculations for backpropagation, given the first training tuple, **X**. The tuple is fed into the network, and the net input and output of each unit are computed. The error of each unit is computed and propagated backward. The error values and the weight and bias updates are calculated. Given learning rate=0.9.

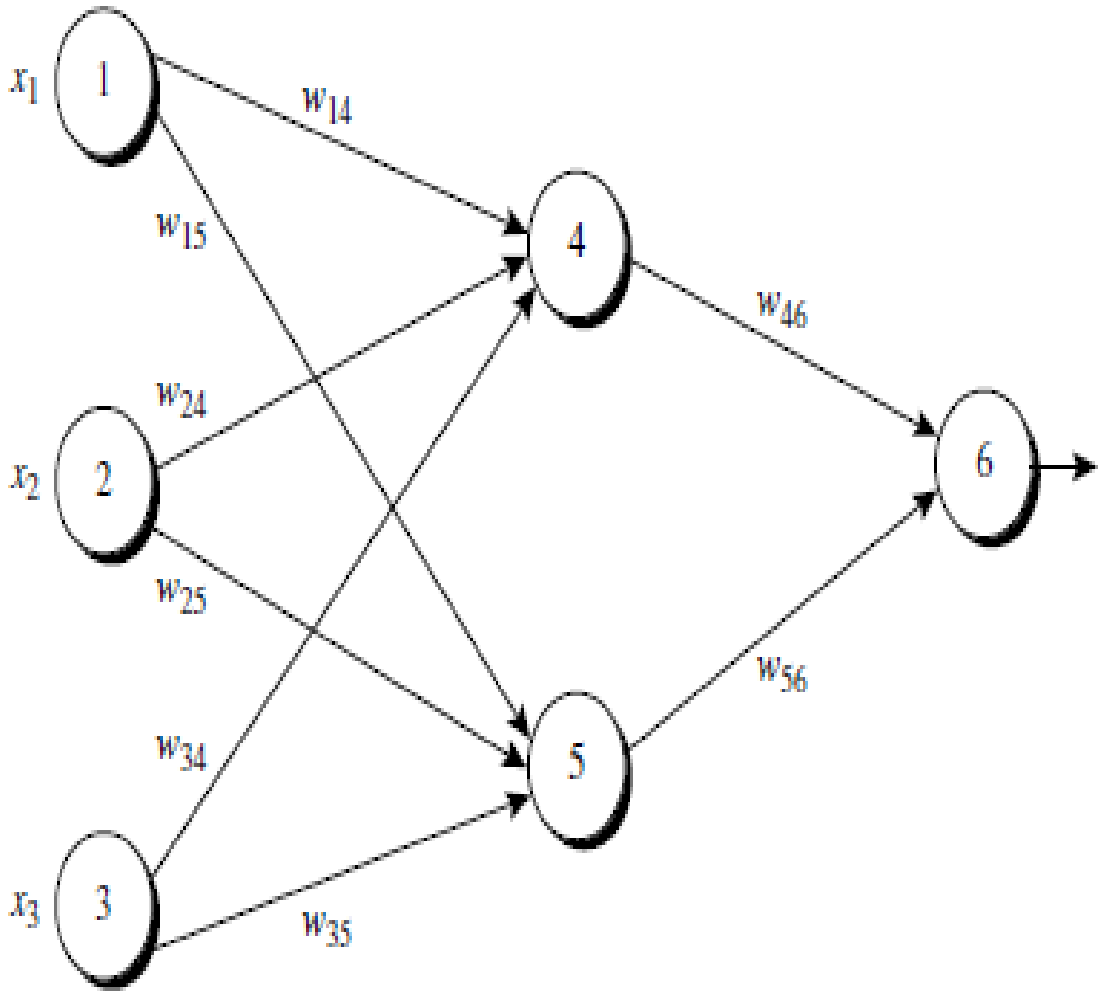


Figure Example of a multilayer feed-forward neural network.

Table .1 Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Table .2 Net Input and Output Calculations

Unit, j	*Net Input*, I_j	*Output*, O_j
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

Table .3 Calculation of the Error at Each Node

Unit, j	Err_j
6	$(0.474)(1-0.474)(1-0.474)=0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2)=-0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3)=-0.0087$

Table .4 Calculations for Weight and Bias Updating

Weight or Bias	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

Training Procedures

Improving Convergence:

- Momentum

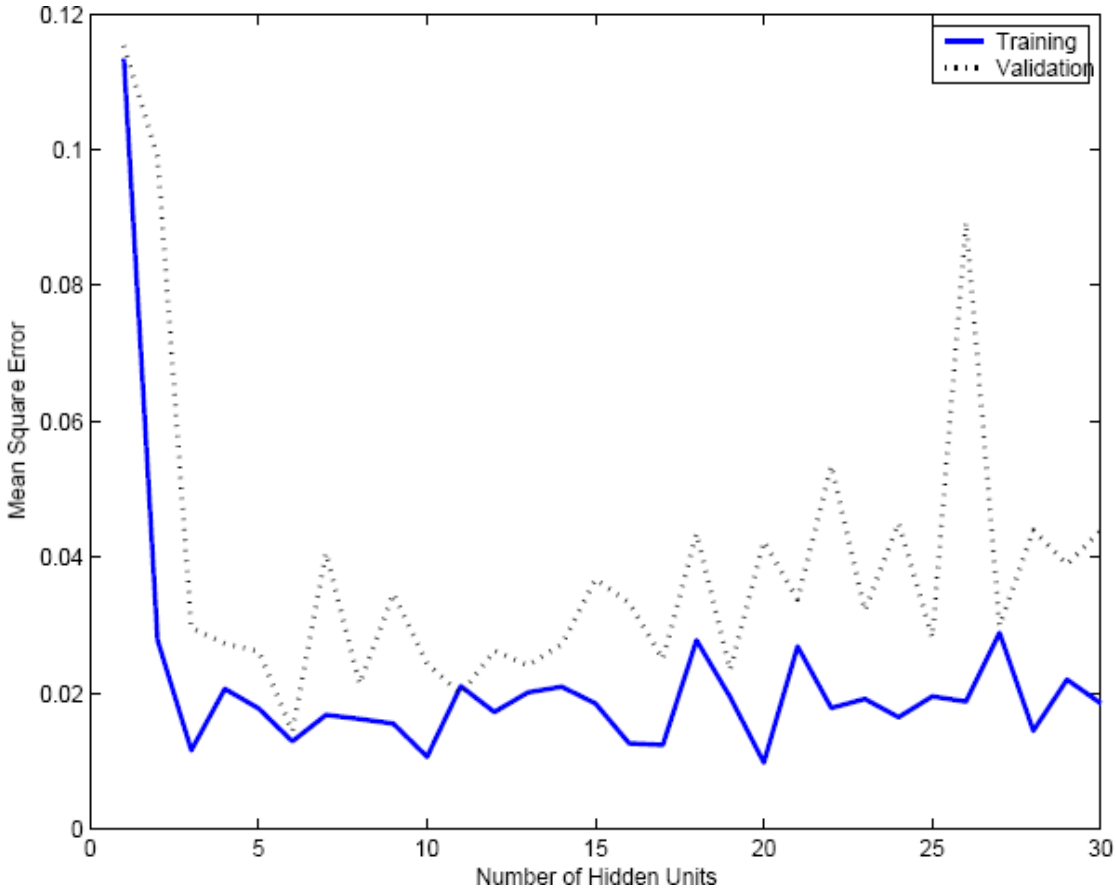
$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

- Adaptive learning rate

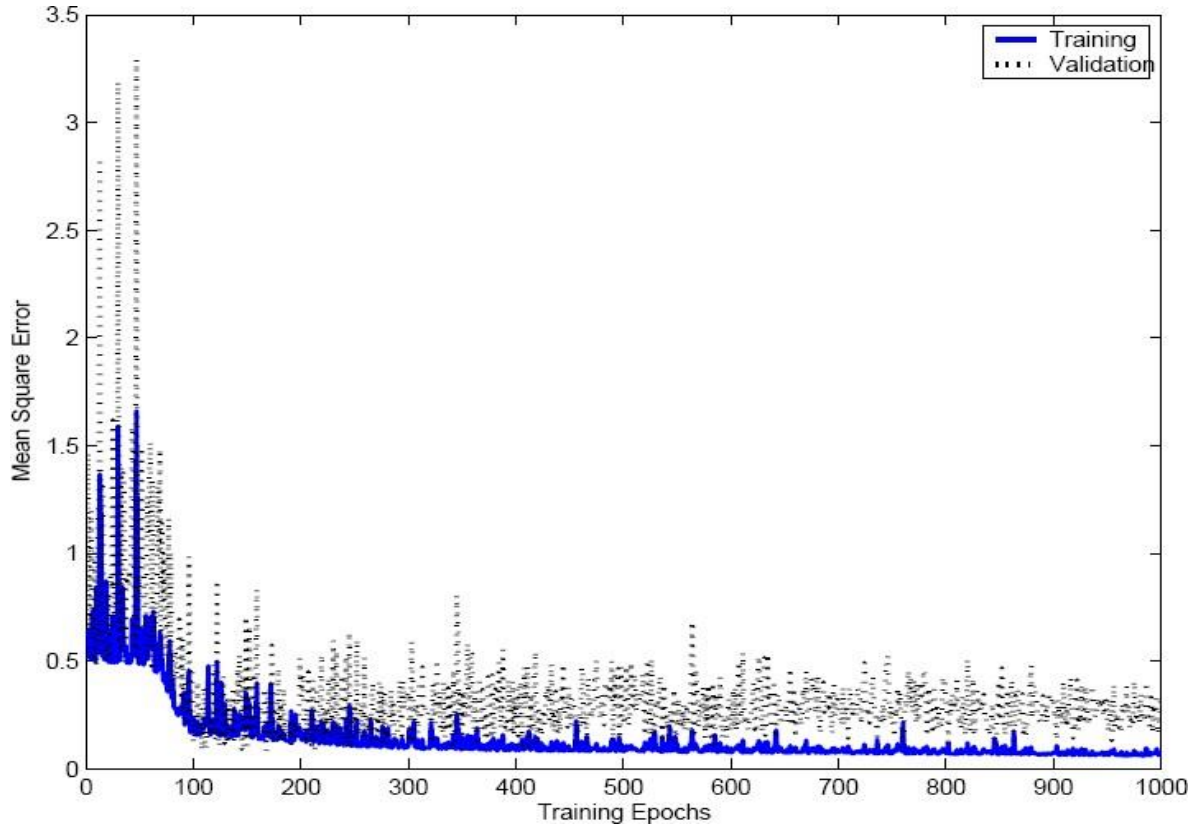
$$\Delta\eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^{t} \\ -b\eta & \text{otherwise} \end{cases}$$

Overfitting/Overtraining:

- Number of weights: H (d+1)+(H+1)K



As complexity increases, training error is fixed but the validation error starts to increase and the network starts to overfit.



As training continues, the validation error starts to increase and the network starts to overfit.

Hints:

- Invariance to translation, rotation, size

- Virtual examples
- Augmented error: $E'=E+\lambda_h E_h$

If $\boldsymbol{x}'$ and $\boldsymbol{x}$ are the "same": $E_h=[g(x|\theta)- g(x'|\theta)]^2$

Approximation hint:

$$E_h=\begin{cases}0 & \text{if } g(x|\theta)\in[a_x,b_x]\\ (g(x|\theta)-a_x)^2 & \text{if } g(x|\theta)<a_x\\ (g(x|\theta)-b_x)^2 & \text{if } g(x|\theta)>b_x\end{cases}$$

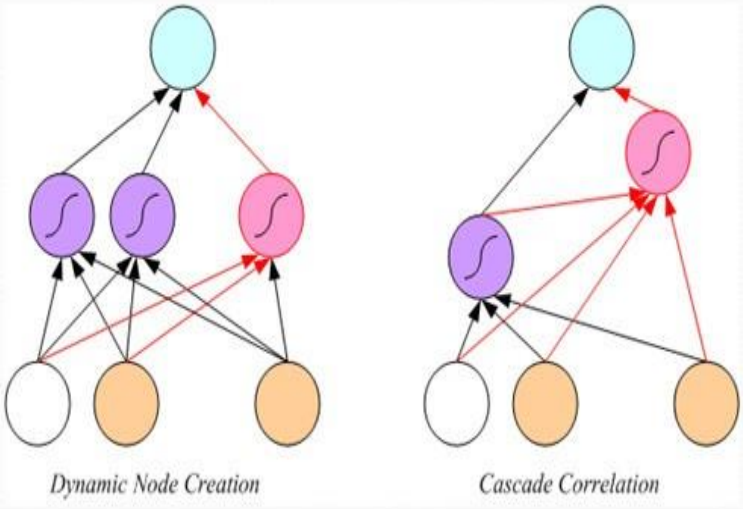
Tuning the Network Size

- Destructive
- Weight decay:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} - \lambda w_i$$

$$E' = E + \frac{\lambda}{2} \sum_i w_i^2$$

- Constructive
- Growing networks



(Ash, 1989) (Fahlman and Lebiere, 1989)

Learning Time

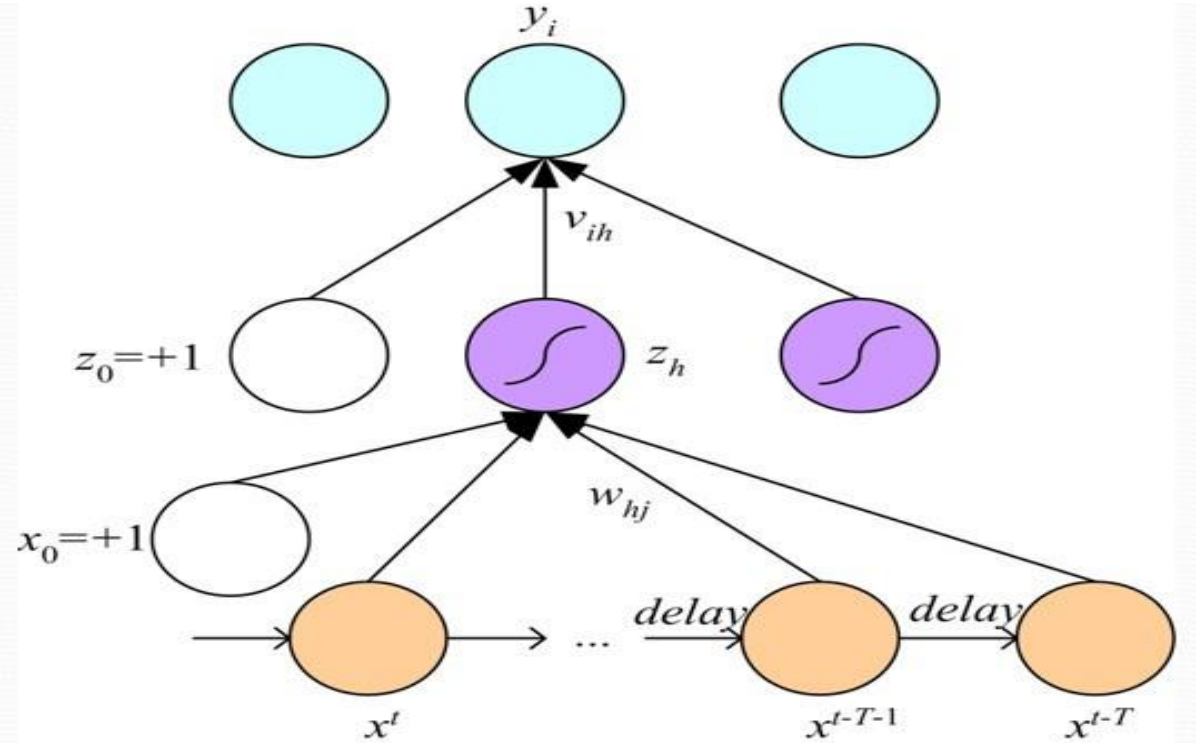
- Applications:
  - Sequence recognition: Speech recognition
  - Sequence reproduction: Time-series prediction
  - Sequence association
- Network architectures
  - Time-delay networks (Waibel et al., 1989)
  - Recurrent networks (Rumelhart et al., 1986)

Time-Delay Neural Networks:

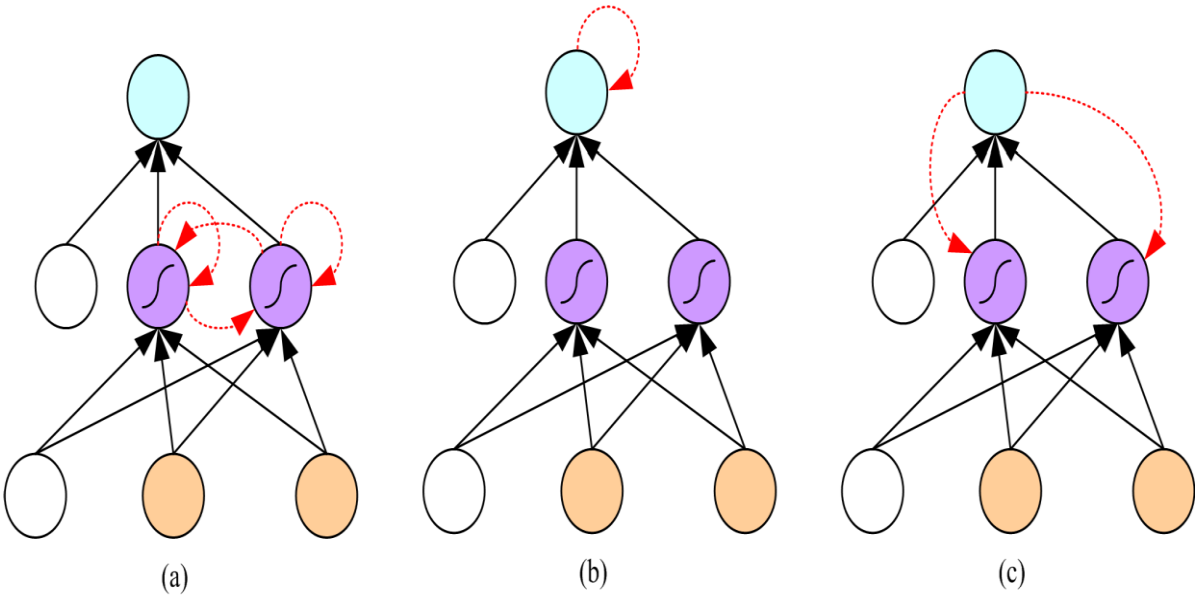
In a time delay neural network, previous inputs are delayed in time so as to synchronize with the final input, and all are fed together as input to the system.Backpropagation can then be

used to train the weights. To extract features local in time, one can have layers of structured connections and weight sharing to get translation invariance in time. The main restriction of this architectureis that he size of the time window we slide over the sequence should be fixed a priori.

Inputs in a time window of length T are delayed in time until we can feed all T inputs as theinput vector to the MLP.



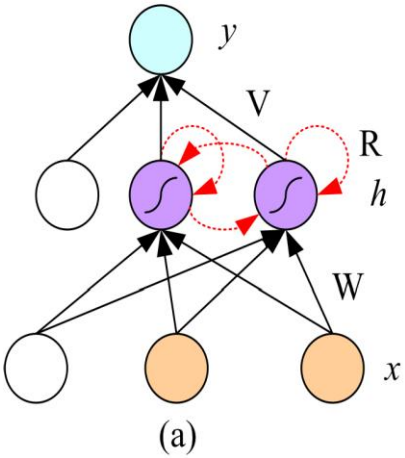
Recurrent Networks:

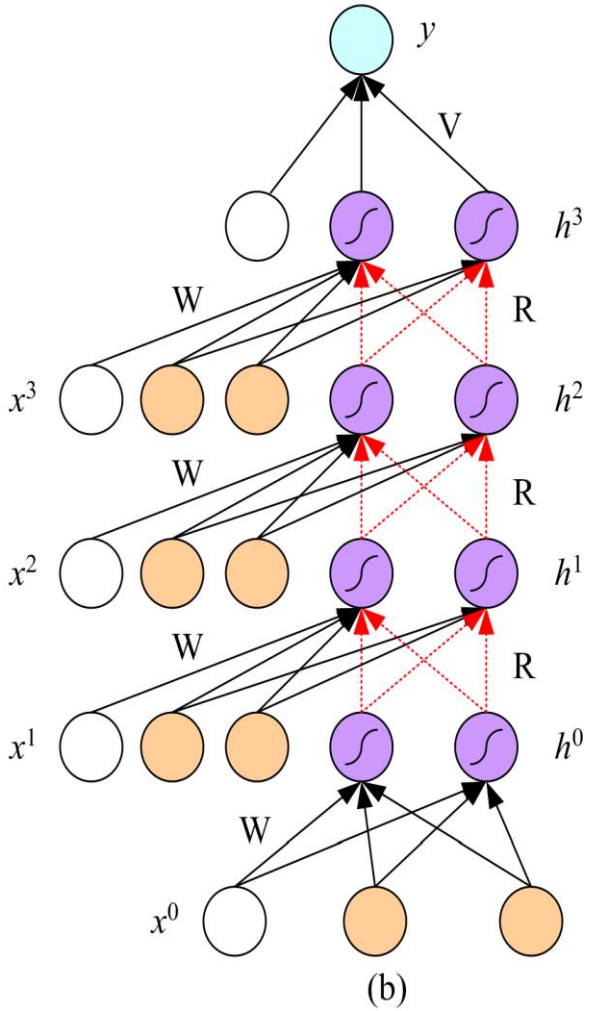


Examples of MLP with partial recurrency. Recurrent connections are shown with dashed lines: (a) self-connections in the hidden layer, (b) selfconnections in the output layer, and (c) connections from the output to the hidden layer. Combinations of these are also possible.

Unfolding in Time:

If the sequences have a small maximum length, then unfolding in time can be used to convert an arbitrary recurrent network to an equivalent feedforward network. A separate unit and connection is created for copies at different times. The resulting network can be trained with backpropagation with the additional requirement that all copies of each connection should remain identical. The solution, as in weight sharing, is to sum up the different weight changes in time and change the weight by the average. This is called backpropagation through time (Rumelhart, Hinton, and Willams 1986b). The problem with this approach is the memory requirement if the length of the sequence is large. Real time recurrent learning (Williams and Zipser 1989) is an algorithm learning for training recurrent networks without unfolding and hasthe advantage that it can use sequences of arbitrary length.





UNIT – V

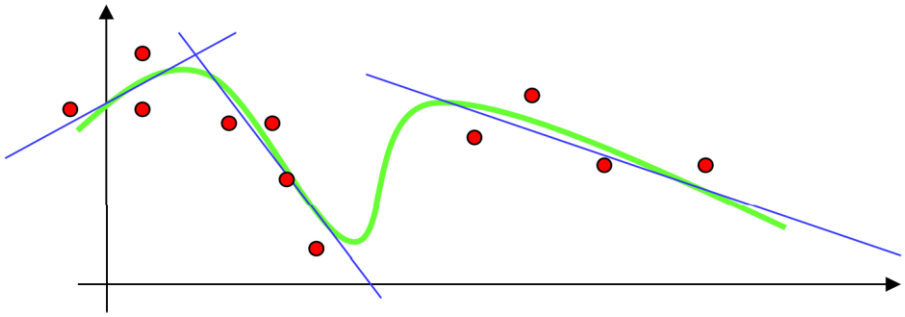
Machine Learning – SCSA1601

UNIT V LOCAL MODELS

Competitive learning - Adaptive resonance theory - Self organizing map – R a d i a l Basis functions – B a g g i n g – B o o s t i n g – Reinforcement Learning.

Introduction

- Divide the input space into local regions and learn simple (constant/linear) models in eachpatch

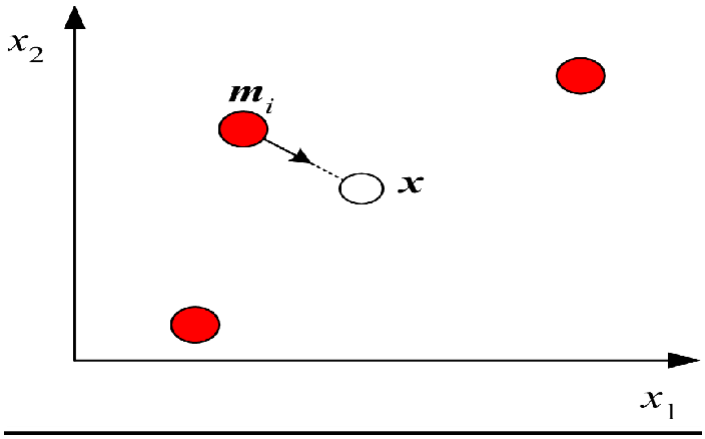


- Unsupervised: Competitive, online clustering
- Supervised: Radial-basis functions, mixture of experts

Competitive learning

The term *competitive learning* is used because it is as if these groups, or rather the units representing these groups, compete among themselves to be the one responsible for representing an instance. The model is also called *winner-take-all*; it is as if one group wins and gets updated, and the others are not updated at all.

An online method has the usual advantages that (1) we do not need extra memory to store the whole training set; (2) updates at each step are simple to implement, for example, in hardware; and (3) the input distribution may change in time and the model adapts itself to these changes automatically. If we were to use a batch algorithm, we would need to collect a new sample and run the batch method from scratch over the whole sample.



Online k-Means:

The reconstruction error is,

$$E\left(\{\mathbf{m}_i\}_{i=1}^{k} \mid X\right) = \sum_t \sum_i b_i^t \left\|\mathbf{x}^t - \mathbf{m}_i\right\|$$

$$b_i^t = \begin{cases} 1 & \text{if } \left\|\mathbf{x}^t - \mathbf{m}_i\right\| = \min_l \left\|\mathbf{x}^t - \mathbf{m}_l\right\| \\ 0 & \text{otherwise} \end{cases}$$

Batch k - means : $\mathbf{m}_i = \dfrac{\sum_t b_i^t \mathbf{x}^t}{\sum_t b_i^t}$

Online k - means :

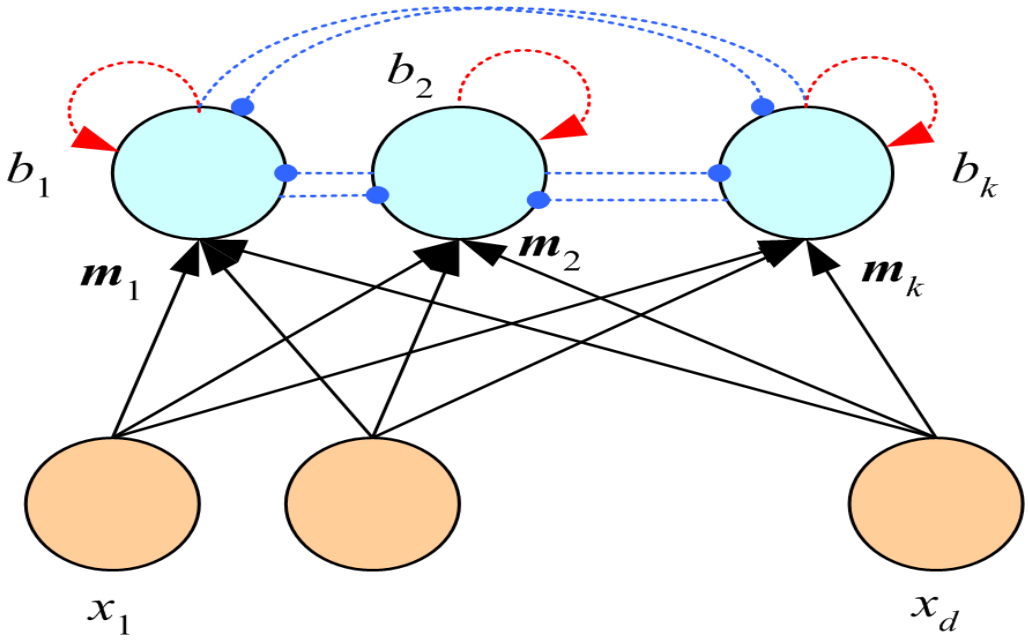
$$\Delta m_{ij} = -\eta \frac{\partial E^t}{\partial m_{ij}} = \eta\, b_i^t \left(x_j^t - m_{ij}\right)$$

Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to k random $\boldsymbol{x}^t$
Repeat
  For all $\boldsymbol{x}^t \in \mathcal{X}$ in random order
    $i \leftarrow \arg\min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\|$
    $\boldsymbol{m}_i \leftarrow \boldsymbol{m}_i + \eta(\boldsymbol{x}^t - \boldsymbol{m}_j)$
Until $\boldsymbol{m}_i$ converge

Online k-means algorithm

Hebbian learning, which defines the update as the product of the values of the presynaptic and postsynaptic units. It was proposed as a model for neural plasticity: A synapse becomes more important if the units before and after the connection fire simultaneously, indicating that they are correlated. However, with only Hebbian learning, the weights grow without bound ($x_j^t \geq 0$), and we need a second force to decrease the weights that are not updated. One possibility is to explicitly normalize the weights to have $\|\boldsymbol{m}_i\| = 1$; if $\Delta m_{ij} > 0$ and $\Delta m_{il} = 0, l \neq i$, once we normalize $\boldsymbol{m}_i$ to unit length, m_{il} decrease.

Winner-take-all network:



The winner-take-all competitive neural network, which is a network of k perceptrons with recurrent connections at the output. Dashed lines are recurrent connections, of which the ones that end with an arrow are excitatory and the ones that end with a circle are inhibitory. Each unit at the output reinforces its value and tries to suppress the other outputs. Under a suitable assignment of these recurrrent weights, the maximum suppresses all the others. This has the net effect that the one unit whose $\boldsymbol{m}_i$ is closest to $\boldsymbol{x}$ ends up with its b_i equal to 1 and all others, namely, $b_l, l \neq i$ are 0.

Adaptive Resonance Theory

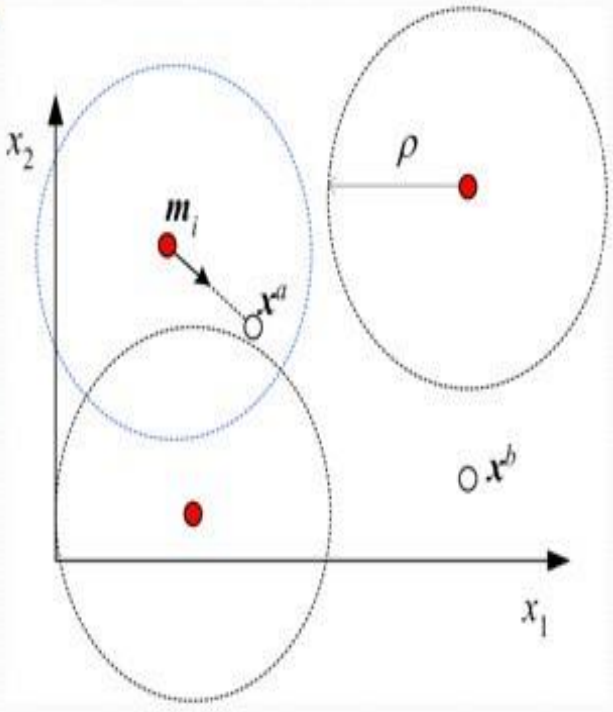
The number of groups, k, should be known and specified before the parameters can be calculated. Another approach is *incremental*, where one starts with a single group and adds new groups as they are needed. We discuss the *adaptive resonance theory* (ART) algorithm (Carpenter and Grossberg 1988) as an example of an incremental algorithm. In ART, given an input, all of the output units calculate their values and the one most similar to the input is chosen. This is the unit with the maximum value if the unit uses the dot product or it is the unit with the minimum value if the unit uses the Euclidean distance.

Let us assume that we use the Euclidean distance. If the minimum value is smaller than a certain threshold value, named the *vigilance*, the update is done as in online k-means. If this distance is larger than vigilance, a new output unit is added and its center is initialized with the instance. This defines a hypersphere whose radius is given by the vigilance defining the volume of scope of each unit; we add a new unit whenever we have an input that is not covered by any unit.

- Incremental; add a new cluster if not covered; defined by vigilance, ρ

$$b_i^t = \left\|\mathbf{x}^t - \mathbf{m}_i\right\| = \min_{l=1}^{k} \left\|\mathbf{x}^t - \mathbf{m}_l\right\|$$

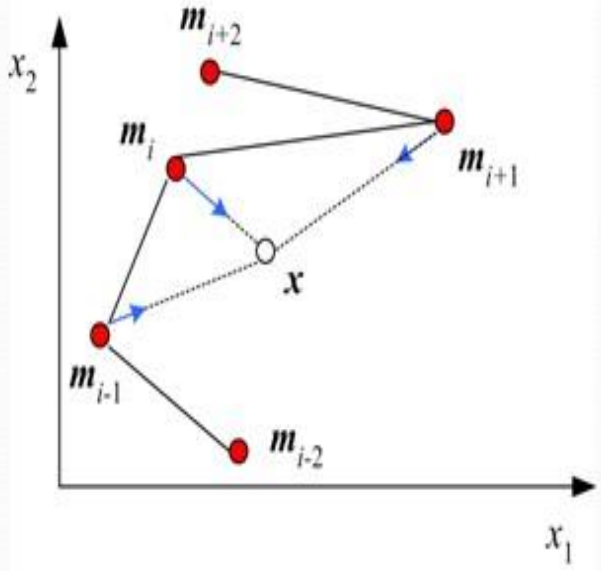
$$\begin{cases} \mathbf{m}_{k+1} \leftarrow \mathbf{x}^t & \text{if } b_i > \rho \\ \Delta\mathbf{m}_i = \eta\left(\mathbf{x}^t - \mathbf{m}_i\right) & \text{otherwise} \end{cases}$$



Self-Organizing Maps

- Units have a neighborhood defined; m_i is "between" m_{i-1} and m_{i+1}, and are all updated together
- One-dim map:

(Kohonen, 1990)



$$\Delta \mathbf{m}_l = \eta e(l,i)\left(\mathbf{x}^t - \mathbf{m}_l\right)$$

$$e(l,i) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left[-\frac{(l-i)^2}{2\sigma^2}\right]$$

In the SOM, not only the closest unit but also its neighbors, in terms of indices, are moved toward the input. Here, neighborhood is 1; m_i and its 1-nearest neighbors are updated. Note here that m_{i+1} is far from m_i, but as it is updated with m_i, and as m_i will be updated when m_{i+1} is the winner, they will become neighbors in the input space as well.

Radial Basis Functions

In a multilayer perceptron where hidden units use the dot product, each hidden unit defines a hyperplane and with the sigmoid nonlinearity, a hidden unit has a value between 0 and 1, coding the position of the instance with respect to the hyperplane. Each hyperplane divides the input space in two, and typically for a given input, many of the hidden units have nonzero output. This is called a *distributed representation* because the input is encoded by the simultaneous activation of many hidden units.

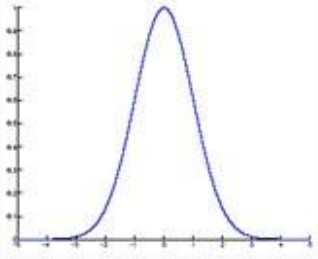
Another possibility is to have a *local representation* where for a given input, only one or a few units are active. It is as if these *locally tuned units* partition the input space among themselves and are selective to only certain inputs. The part of the input space where a unit has nonzero response is called its *receptive field*. The input space is then paved with such units.

Neurons with such response characteristics are found in many parts of the cortex. For example, cells in the visual cortex respond selectively to stimulation that is both local in retinal position and local in angle of visual orientation. Such locally tuned cells are typically arranged in topogrophical cortical maps in which the values of the variables to which the cells respond vary by their position in the map, as in a SOM.

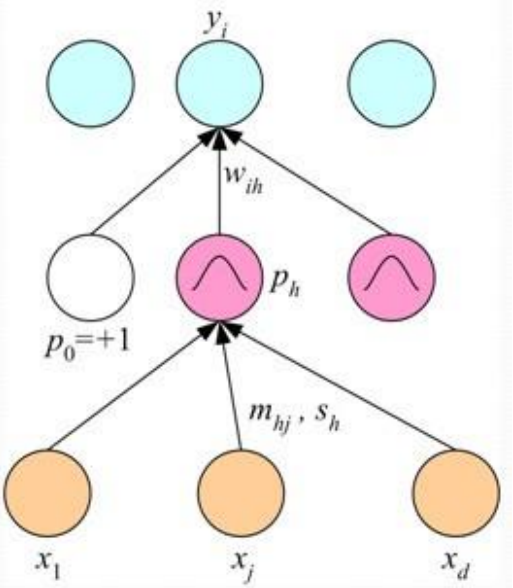
The concept of locality implies a distance function to measure the similarity between the given input $\boldsymbol{x}$ and the position of unit h, $\boldsymbol{m}_h$. Frequently this measure is taken as the Euclidean distance, $\|\boldsymbol{x} - \boldsymbol{m}_h\|$. The response function is chosen to have a maximum where $\boldsymbol{x} = \boldsymbol{m}_h$ and decreasing as they get less similar.

- Locally-tuned units:

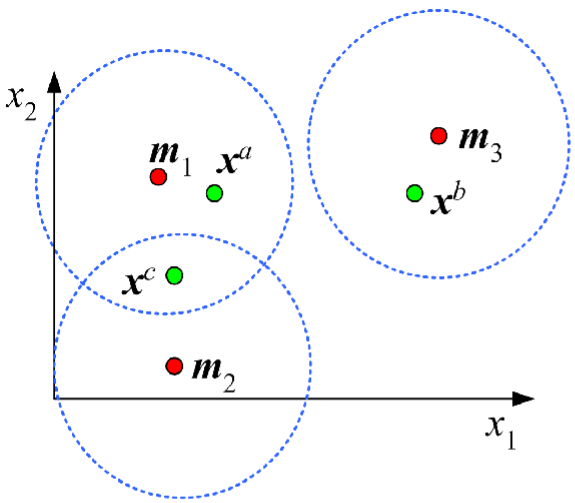
$$p_h^t = \exp\left[-\frac{\left\|\mathbf{x}^t - \mathbf{m}_h\right\|^2}{2s_h^2}\right]$$

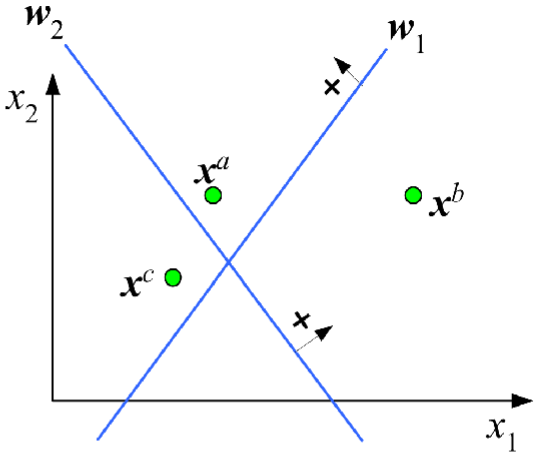


$$y^t = \sum_{h=1}^{H} w_h p_h^t + w_0$$



Local vs Distributed Representation:





Local representation in the space of (p_1, p_2, p_3)	Distributed representation in the space of (h_1, h_2)
x^a: (1.0, 0.0, 0.0)	x^a: (1.0, 1.0)
x^b: (0.0, 0.0, 1.0)	x^b: (0.0, 1.0)
x^c: (1.0, 1.0, 0.0)	x^c: (1.0, 0.0)

The difference between local and distributed representations.

The values are hard, 0/1, values. One can use soft values in (0, 1) and get a more informative encoding. In the local representation, this is done by the Gaussian RBF that uses the distance to the center, $\boldsymbol{m}_i$, and in the distributed representation, this is done by the sigmoid that uses the distance to the hyperplane, $\boldsymbol{w}_i$.

Regression:

$$E\left(\{\mathbf{m}_h, s_h, w_{ih}\}_{i,h} \mid X\right) = \frac{1}{2}\sum_t \sum_i \left(r_i^t - y_i^t\right)^2$$

$$y_i^t = \sum_{h=1}^{H} w_{ih} p_h^t + w_{i0}$$

$$\Delta w_{ih} = \eta \sum_t \left(r_i^t - y_i^t\right) p_h^t$$

$$\Delta m_{hj} = \eta \sum_t \left[\sum_i \left(r_i^t - y_i^t\right) w_{ih}\right] p_h^t \frac{\left(x_j^t - m_{hj}\right)}{s_h^2}$$

$$\Delta s_h = \eta \sum_t \left[\sum_i \left(r_i^t - y_i^t\right) w_{ih}\right] p_h^t \frac{\left\|\mathbf{x}^t - \mathbf{m}_h\right\|^2}{s_h^3}$$

Training RBF:

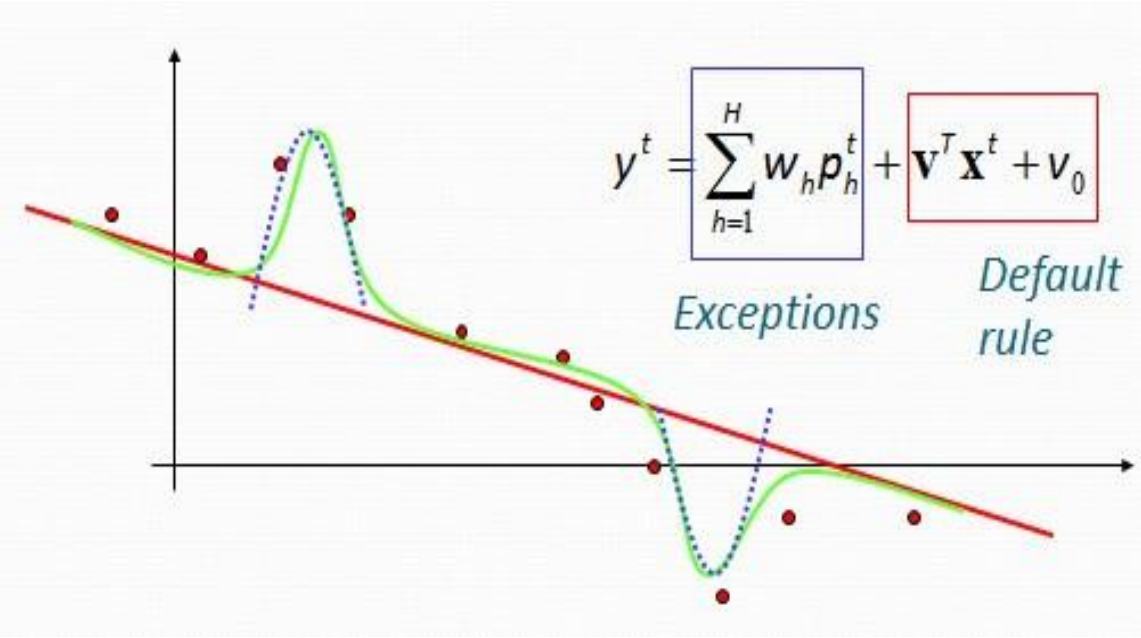
- Hybrid learning:
  - First layer centers and spreads:
    Unsupervised *k*-means
  - Second layer weights:
    Supervised gradient-descent
- Fully supervised

Classification:

$$E\left(\{\mathbf{m}_h, s_h, w_{ih}\}_{i,h} \mid X\right) = -\sum_t \sum_i r_i^t \log y_i^t$$

$$y_i^t = \frac{\exp\left[\sum_h w_i \; p_h^t + w_{i0}\right]}{\sum_k \exp\left[\sum_h w_k \; p_h^t + w_{k0}\right]}$$

Rules and Exceptions:



Learning Vector Quantization

Let us say we have H units for each class, already labeled by those classes. These units are initialized with random instances from their classes. At each iteration, we find the unit, $\mathbf{m}_i$, that is closest to the input instance in Euclidean distance and use the following update rule:

$$\begin{cases} \Delta \mathbf{m}_i = \eta(\mathbf{x}^t - \mathbf{m}_i) & \text{if } \mathbf{x}^t \text{ and } \mathbf{m}_i \text{ have the same class label} \\ \Delta \mathbf{m}_i = -\eta(\mathbf{x}^t - \mathbf{m}_i) & \text{otherwise} \end{cases}$$

If the closest center has the correct label, it is moved toward the input to better represent it. If it belongs to the wrong class, it is moved away from the input in the expectation that if it is moved sufficiently away, a center of the correct class will be the closest in a future iteration. This is the *learning vector quantization* (LVQ) model proposed by Kohonen (1990, 1995).

The LVQ update equation is analogous to

$$\Delta m_{hj} = \eta \sum_t (f_h^t - g_h^t) \frac{(x_j^t - m_{hj})}{s_h^2}$$

where the

direction in which the center is moved depends on the difference between two values: our prediction of the winner unit based on the input distances and what the winner should be based on the required output.

Assessing and Comparing Classification Algorithms

$$L\left(\{\mathbf{m}_h, s_h, w_{ih}\}_{i,h} \mid X\right) = \sum_t \log \sum_h g_h^t \prod_i \left(y_{ih}^t\right)^{r_i^t}$$

$$= \sum_t \log \sum_h g_h^t \exp\left[\sum_i r_i^t \log y_{ih}^t\right]$$

$$y_{ih}^t = \frac{\exp w_{ih}}{\sum_k \exp w_{kh}} = \frac{\exp \mathbf{v}_{ih} \mathbf{x}^t}{\sum_k \exp \mathbf{v}_k \mathbf{x}^t}$$

Combining Multiple Learners

- Early integration: Concat all features and train a single learner
- Late integration: With each feature set, train one learner, then either use a fixed rule or stacking to combine decisions
- Intermediate integration: With each feature set, calculate a kernel, then use a single SVM with multiple kernels
- Combining features vs decisions vs kernels

Rationale:

- No Free Lunch Theorem: There is no algorithm that is always the most accurate
- Generate a group of base-learners which when combined has higher accuracy
- Different learners use different
  - Algorithms
  - Hyperparameters
  - Representations /Modalities/Views
  - Training sets
  - Subproblems
- Diversity vs accuracy

Voting:

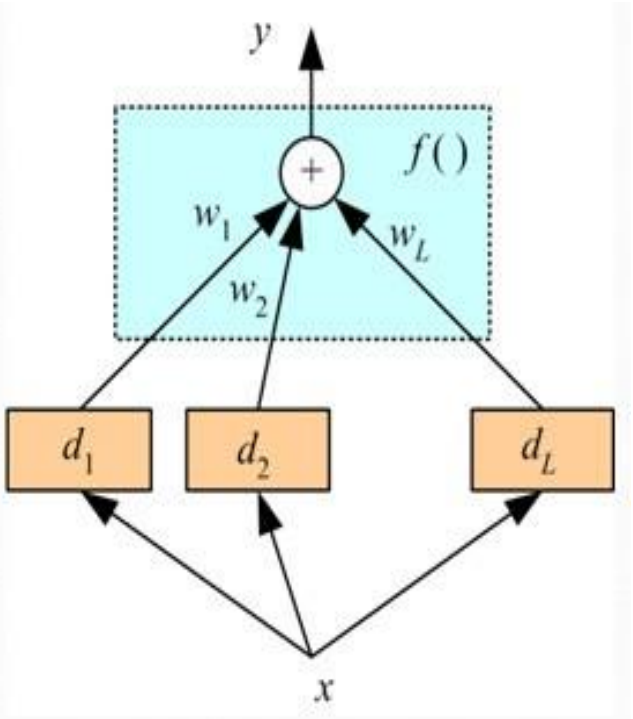
- Linear combination

$$y = \sum_{j=1}^{L} w_j d_j$$

$$w_j \geq 0 \text{ and } \sum_{j=1}^{L} w_j = 1$$

- Classification

$$y_i = \sum_{j=1}^{L} w_j d_{ji}$$



- Bayesian perspective:

$$P(C_i \mid x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i \mid x, \mathcal{M}_j) P(\mathcal{M}_j)$$

If d_j are iid

$$E[y] = E\left[\sum_j \frac{1}{L} d_j\right] = \frac{1}{L} L \cdot E[d_j] = E[d_j]$$

$$\text{Var}(y) = \text{Var}\left(\sum_j \frac{1}{L} d_j\right) = \frac{1}{L^2} \text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2} L \cdot \text{Var}(d_j) = \frac{1}{L} \text{Var}(d_j)$$

Bias does not change, variance decreases by L

- If dependent, error increase with positive correlation

$$\text{Var}(y) = \frac{1}{L^2} \text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \text{Var}(d_j) + 2\sum_j \sum_{i<j} Cov(d_i, d_j)\right]$$

Fixed Combination Rules:

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

Error-Correcting Output Codes:

- K classes; L problems (Dietterich and Bakiri, 1995)
- Code matrix **W** codes classes in terms of learners
- One per class
  $L=K$

$$\mathbf{W}=\begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

- Pairwise
  $L=K(K-1)/2$

$$\mathbf{W}=\begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

- Full code $L=2^{(K-1)}-1$

$$\mathbf{W}=\begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

- With reasonable L, find **W** such that the Hamming distance btw rows and columns are maximized.
- Voting scheme

$$y_i = \sum_{j=1}^{L} w_j d_{ji}$$

- Subproblems may be more difficult than one-per-K

Bagging:

- Use bootstrapping to generate L training sets and train one base-learner with each(Breiman, 1996)
- Use voting (Average or median with regression)
- Unstable algorithms profit from bagging

AdaBoost:

Generate a sequence of base-learners each focusing on previous one's errors

(Freund and Schapire, 1996).

Training:

For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$

For all base-learners $j = 1, \ldots, L$

Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities p_j^t

Train d_j using $\mathcal{X}_j$

For each (x^t, r^t), calculate $y_j^t \leftarrow d_j(x^t)$

Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$

If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop

$\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$

For each (x^t, r^t), decrease probabilities if correct:

If $y_j^t = r^t$ $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$

Normalize probabilities:

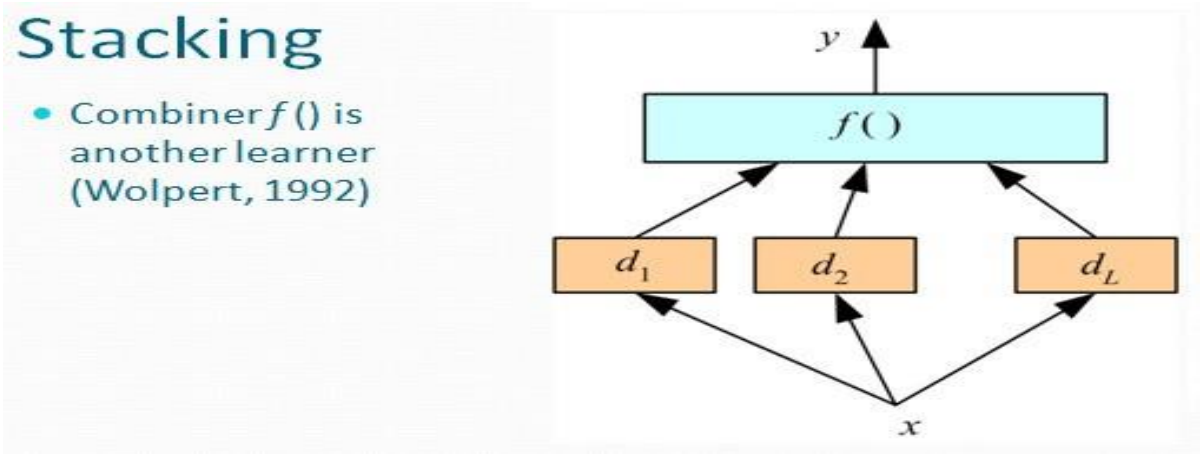
$Z_j \leftarrow \sum_t p_{j+1}^t$; $p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$

Testing:

Given x, calculate $d_j(x), j = 1, \ldots, L$

Calculate class outputs, $i = 1, \ldots, K$:

$$y_i = \sum_{j=1}^{L} \left(\log \frac{1}{\beta_j} \right) d_{ji}(x)$$

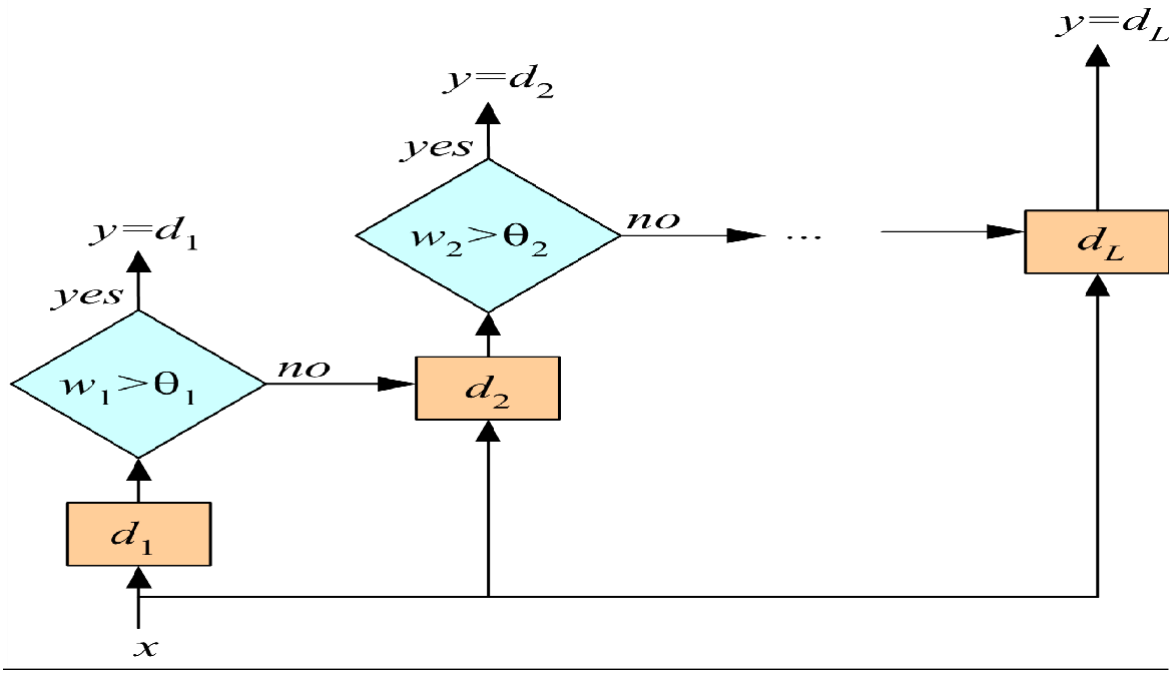


Fine-Tuning an Ensemble:

- Given an ensemble of dependent classifiers, do not use it as is, try to get independence
  1. Subset selection: Forward (growing)/Backward (pruning) approaches to improve accuracy/diversity/independence
  2. Train metaclassifiers: From the output of correlated classifiers, extract new combinations that are uncorrelated. Using PCA, we get "eigenlearners."
- Similar to feature selection vs feature extraction

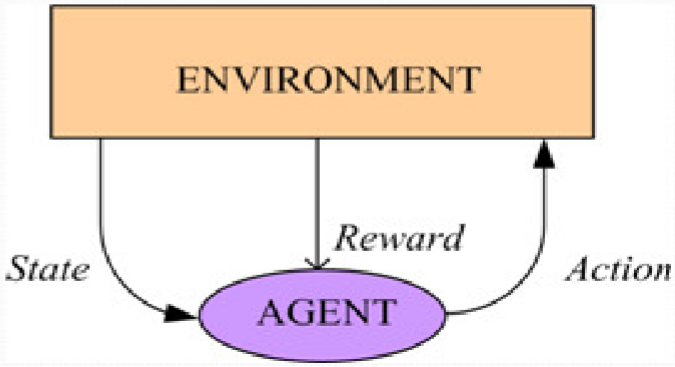
Cascading:

- Use d_j only if preceding ones are not confident
- Cascade learners in order of complexity



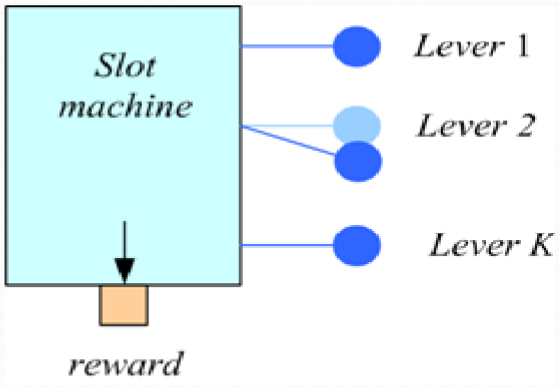
Reinforcement Learning

- Game-playing: Sequence of moves to win a game
- Robot in a maze: Sequence of actions to find a goal
- Agent has a state in an environment, takes an action and sometimes receives reward and the state changes
- Credit-assignment
- Learn a policy



Single State: K-armed Bandit

- Among K levers, choose the one that pays best
  - $Q(a)$: value of action a
  - Reward is r_a
  - Set $Q(a) = r_a$
  - Choose a^* if $Q(a^*)=\max_a Q(a)$



- Rewards stochastic (keep an *expected* reward):

$$Q_{t+1}(a) \leftarrow Q_t(a) + \eta\left[r_{t+1}(a) - Q_t(a)\right]$$

Elements of RL (Markov Decision Processes):

- s_t : State of agent at time t
- a_t: Action taken at time t
- In s_t, action a_t is taken, clock ticks and reward r_{t+1} is received and state changes to s_{t+1}

- Next state prob: $P(s_{t+1} \mid s_t, a_t)$
- Reward prob: $p(r_{t+1} \mid s_t, a_t)$
- Initial state(s), goal state(s)
- Episode (trial) of actions from initial state to goal
- (Sutton and Barto, 1998; Kaelbling et al., 1996)

Policy and Cumulative Reward:

- Policy, $\pi : S \to \mathcal{A} \quad a_t = \pi(s_t)$
- Value of a policy, $V^{\pi}(s_t)$
- Finite-horizon:

$$V^{\pi}(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]$$

- Infinite horizon:

$$V^{\pi}(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]$$

$0 \le \gamma < 1$ is the discount rate

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t$$

$$= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]$$

$$= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right]$$

$$= \max_{a_t} E[r_{t+1} + \gamma V^*(s_{t+1})]$$ Bellman's equation

$$V^*(s_t) = \max_{a_t}\left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) V^*(s_{t+1})\right)$$

$$V^*(s_t) = \max_{a_t} Q^*(s_t, a_t)$$ Value of a_t in s_t

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

Model-Based Learning:

- Environment, $P(s_{t+1} \mid s_t, a_t)$, $p(r_{t+1} \mid s_t, a_t)$, is known
- There is no need for exploration
- Can be solved using dynamic programming
- Solve for

$$V^*(s_t) = \max_{a_t}\left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) V^*(s_{t+1})\right)$$

- Optimal policy

$$\pi^*(s_t) = \arg\max_{a_t}\left(E[r_{t+1} \mid s_t, a_t] + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) V^*(s_{t+1})\right)$$

Value Iteration:

Initialize $V(s)$ to arbitrary values
Repeat
  For all $s \in \mathcal{S}$
    For all $a \in \mathcal{A}$
      $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s')$
    $V(s) \leftarrow \max_a Q(s,a)$
Until $V(s)$ converge

Policy Iteration:

Initialize a policy π arbitrarily
Repeat
  $\pi \leftarrow \pi'$
  Compute the values using π by
    solving the linear equations
      $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V^{\pi}(s')$
  Improve the policy at each state
    $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$

Q-learning

Initialize all $Q(s, a)$ arbitrarily
For all episodes
  Initalize s
  Repeat
    Choose a using policy derived from Q, e.g., ϵ-greedy
    Take action a, observe r and s'
    Update $Q(s, a)$:
      $Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
  Until s is terminal state

Sarsa:

Initialize all $Q(s,a)$ arbitrarily

For all episodes

  Initalize s

  Choose a using policy derived from Q, e.g., ϵ-greedy

  Repeat

    Take action a, observe r and s'

    Choose a' using policy derived from Q, e.g., ϵ-greedy

    Update $Q(s,a)$:

      $$Q(s,a) \leftarrow Q(s,a) + \eta(r + \gamma Q(s',a') - Q(s,a))$$

    $s \leftarrow s',\ a \leftarrow a'$

  Until s is terminal state