Agenda

Day 3	1. Instance Based Learning: Introduction, K-nearest neighbor learning, locally weighted regression, radial basis function, cased based reasoning. 2. Reinforced Learning: Introduction, Learning Task, Q Learning

- https://github.com/profthyagu
- http://proftgs.blogspot.com
- Gmail1: profthyagu@gmail.com
- Gmal2 : innovationscontext@gmail.com
- **Mobile** : 9480123526

1.1 Instance Based Learning

- **Memory-based learning** (also called **instance-based learning**) is a type of learning algorithm that compares new test data with training data in order to solve the given machine learning problem.
- Such algorithms **search for the training data that are most similar to the test data** and make predictions based on these similarities.
- Learning in these algorithms consists of simply storing the presented training data. ***When a new query instance is encountered a set of similar related instances is retrieved from memory and used to classify the new query instance.***
- **Examples :** *k-nearest neighbor learning , locally weighted regression,* ***Radial Basis function (RBF),*** *kernel machines and Case based Reasoning .*

Key idea of Instance based learning

- **Key idea**: ***instance-based learning constructs the target function only when a new instance must be classified***.
- Only store all training examples $< x_i, f(x_i) >$ where *x* describes the attributes of each instance and *f(x)* denotes ***its class (or value)***.
- **Use a Nearest Neighbor method:**

  Given query instance x_q ,

  first locate nearest (most similar) training example x_n ,

  then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

K Nearest Neighbor Learning

- The nearest neighbours of an instance are defined in terms of the standard Euclidean Distance ***d*(*x*$_i$, *x*$_j$)** .

- Let an arbitrary instance x be described by the feature vector (set of attributes) as follows:

$$< a_1(x), a_2(x), \ldots a_n(x) >$$

where ***a*$_r$(*x*)** denotes the value of the **r**th attribute of instance ***x***. Then the distance between two instances **x**$_i$ and **x**$_j$ is defined to be ***d*(*x*$_i$, *x*$_j$)** where

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

K Nearest Neighbor Algorithm

Training Algorithm

- For each **training example** ***<x , f(x) >,*** add the example to the list of *training_examples*

Classification Algorithm

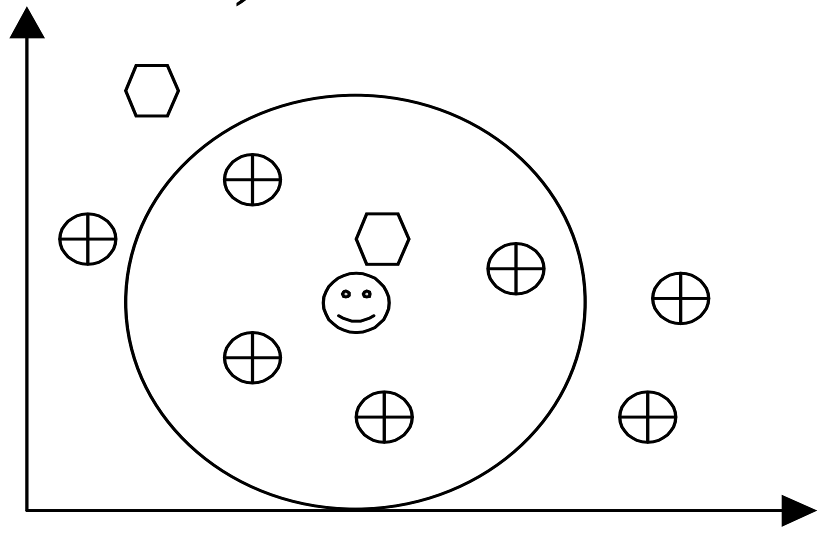
- Given a query instance x_q to be classified
    - **Let $x_1 \dots x_k$ denote the k instances from *training_examples* that are nearest to x_q**
    - **Return**

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

$$\text{where } \delta(\mathrm{a},\mathrm{b}) = 1 \quad \text{if } a = b$$
$$= 0 \quad \text{otherwise}$$

k-Nearest Neighbor:

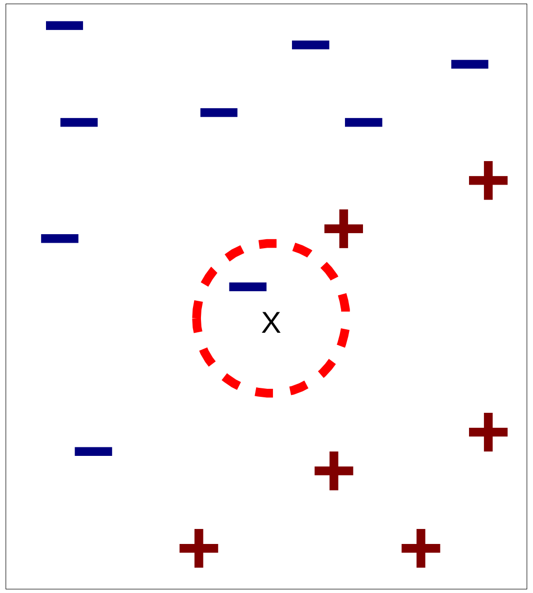
Given query instance x_q , take **vote** among its *k* nearest neighbors to decide its class, (return most common value of f among the k nearest training elements to x_q)

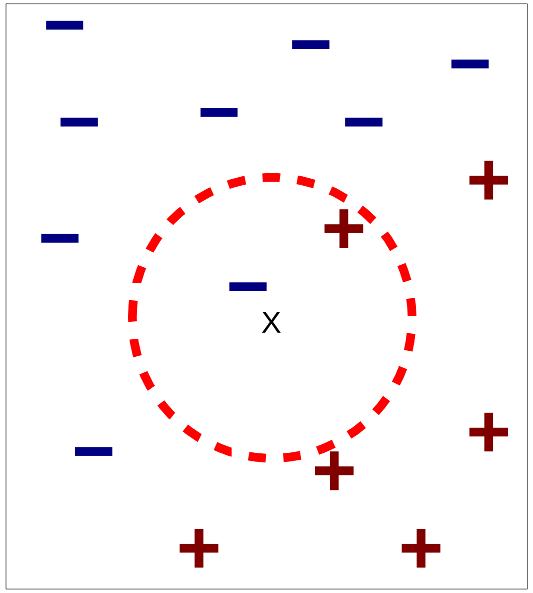


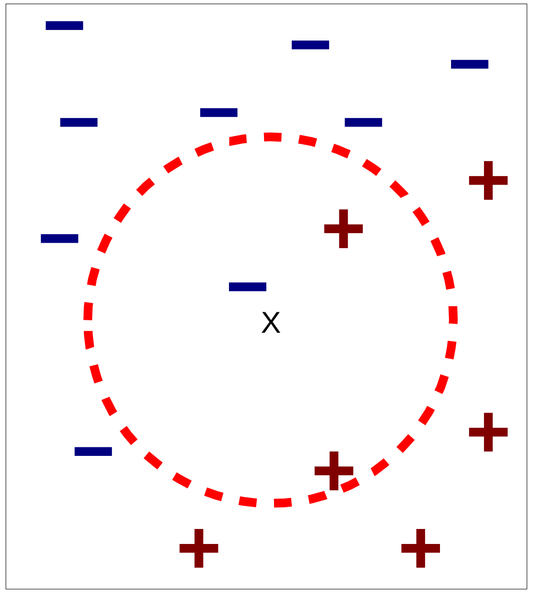
1-NN, class is ⬡,   5-NN, class is ⊕

Advantage: overcome class noise in training set.

Example







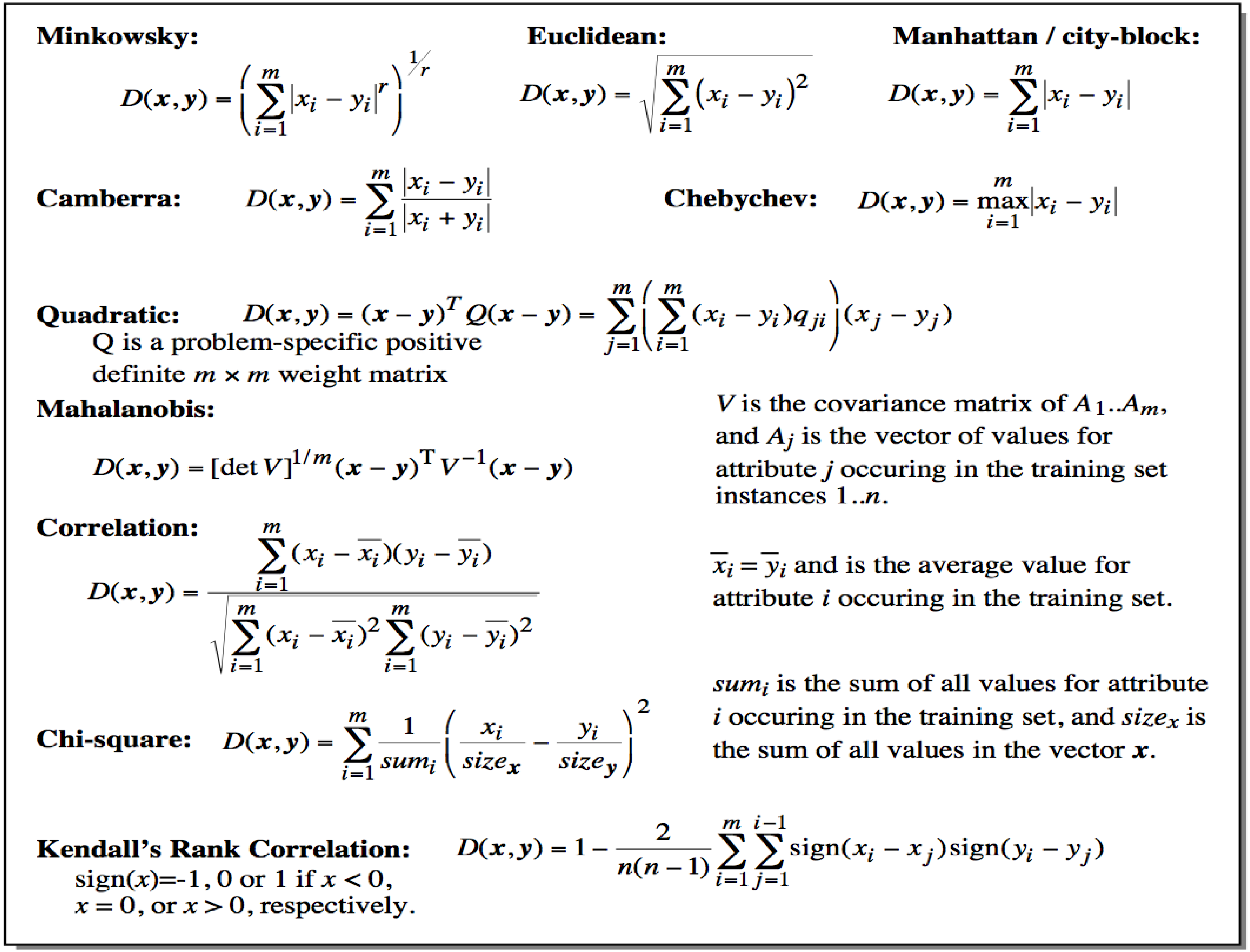
(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

Continuous vs Discrete valued functions (classes)

K-NN works well for discrete-valued target functions. Furthermore, the idea can be extended f or continuos (real) valued functions. In this case we can take mean of the f values of k nearest neighbors:

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Distance Metrics



Minkowsky:

$$D(\boldsymbol{x},\boldsymbol{y}) = \left(\sum_{i=1}^{m} |x_i - y_i|^r \right)^{1/r}$$

Euclidean:

$$D(\boldsymbol{x},\boldsymbol{y}) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

Manhattan / city-block:

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} |x_i - y_i|$$

Camberra:

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} \frac{|x_i - y_i|}{|x_i + y_i|}$$

Chebychev:

$$D(\boldsymbol{x},\boldsymbol{y}) = \max_{i=1}^{m} |x_i - y_i|$$

Quadratic:

$$D(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x} - \boldsymbol{y})^T Q(\boldsymbol{x} - \boldsymbol{y}) = \sum_{j=1}^{m} \left(\sum_{i=1}^{m} (x_i - y_i) q_{ji} \right) (x_j - y_j)$$

Q is a problem-specific positive definite $m \times m$ weight matrix

Mahalanobis:

$$D(\boldsymbol{x},\boldsymbol{y}) = [\det V]^{1/m} (\boldsymbol{x} - \boldsymbol{y})^{\mathrm{T}} V^{-1} (\boldsymbol{x} - \boldsymbol{y})$$

V is the covariance matrix of $A_1..A_m$, and A_j is the vector of values for attribute j occuring in the training set instances $1..n$.

Correlation:

$$D(\boldsymbol{x},\boldsymbol{y}) = \frac{\sum_{i=1}^{m} (x_i - \overline{x_i})(y_i - \overline{y_i})}{\sqrt{\sum_{i=1}^{m} (x_i - \overline{x_i})^2 \sum_{i=1}^{m} (y_i - \overline{y_i})^2}}$$

$\overline{x_i} = \overline{y_i}$ and is the average value for attribute i occuring in the training set.

Chi-square:

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} \frac{1}{sum_i} \left(\frac{x_i}{size_x} - \frac{y_i}{size_y} \right)^2$$

sum_i is the sum of all values for attribute i occuring in the training set, and $size_x$ is the sum of all values in the vector $\boldsymbol{x}$.

Kendall's Rank Correlation:

$$D(\boldsymbol{x},\boldsymbol{y}) = 1 - \frac{2}{n(n-1)} \sum_{i=1}^{m} \sum_{j=1}^{i-1} \operatorname{sign}(x_i - x_j) \operatorname{sign}(y_i - y_j)$$

sign(x)=-1, 0 or 1 if $x < 0$, $x = 0$, or $x > 0$, respectively.

Figure 1. Equations of selected distance functions. ($\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of m attribute values).

When To Consider Nearest Neighbor ?

- Instances map to points in $\Re^n$
- Average number of attributes (e.g. Less than 20 attributes per instance)
- Lots of training data
- When target function is complex but can be approximated by separate local simple approximations

Advantages:	**Disadvantages:**
Training is very fast	Slow at query time
Learn complex target functions	Easily fooled by irrelevant attributes

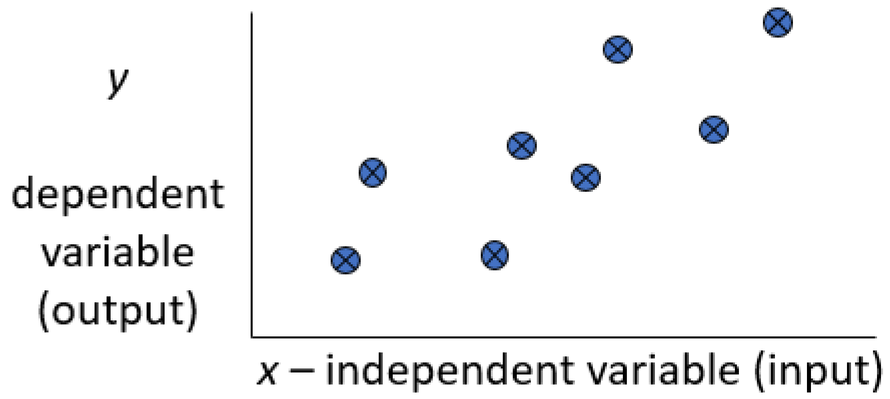
Lab Program9

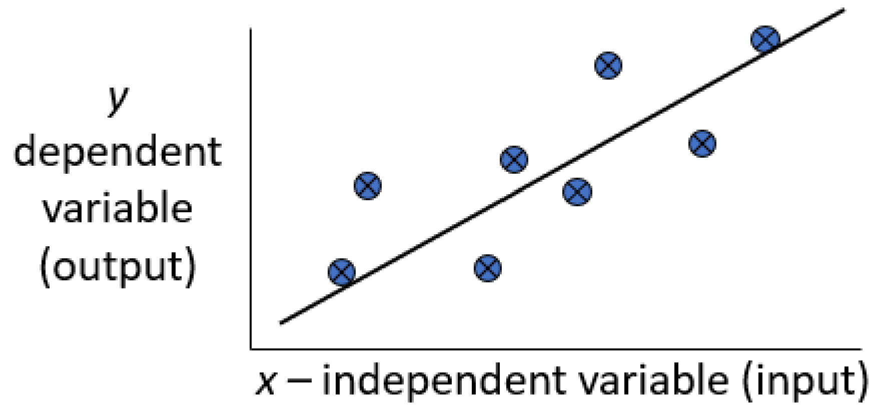
- Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Python ML library classes can be used for this problem.

Source Code

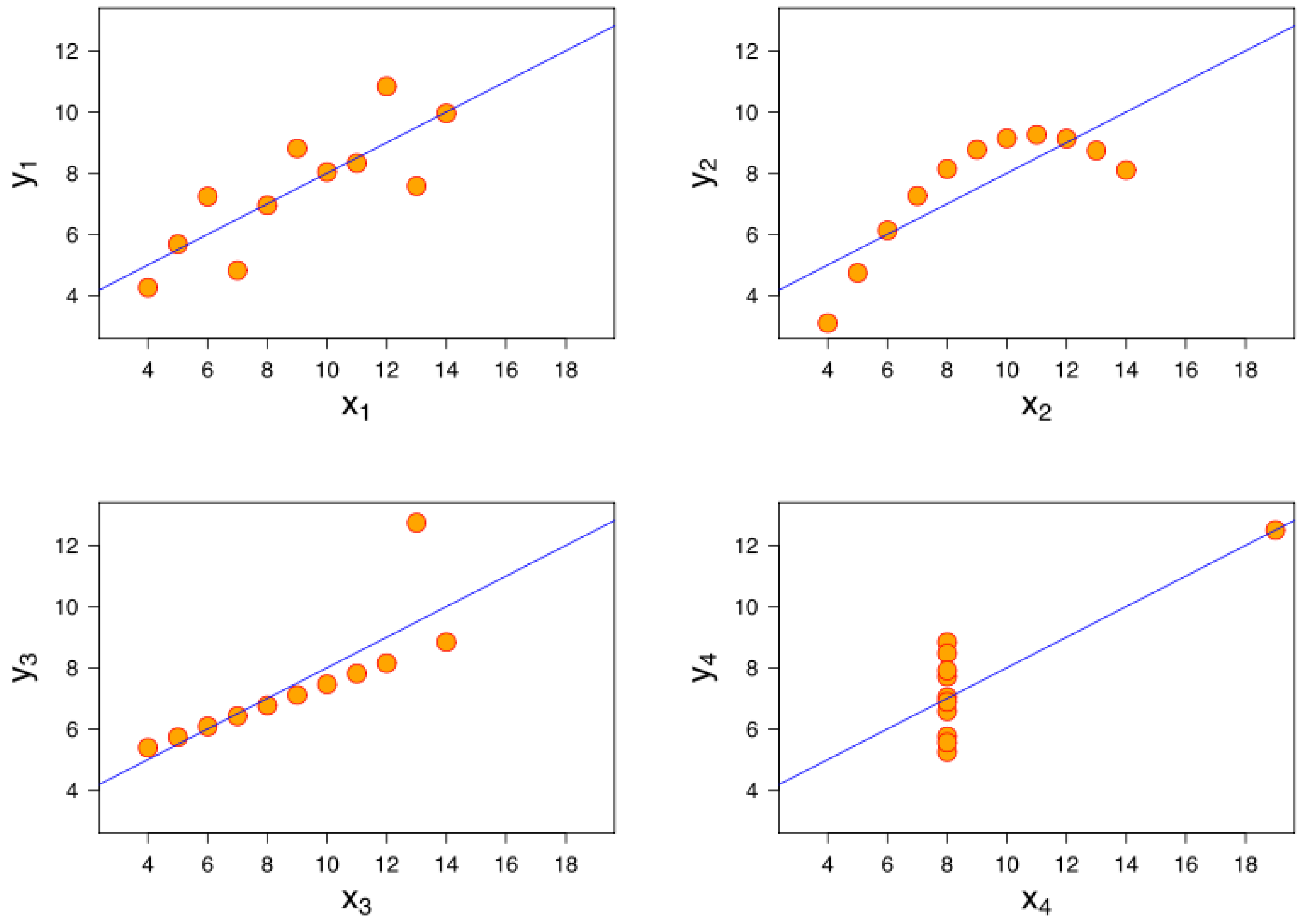
Regression

- Regression is a technique from statistics that is used to predict values of a desired target quantity when the ***target quantity is continuous*** .
- In regression, we seek to ***identify (or estimate)*** a continuous variable y associated with a given input vector x.
  - y is called the **dependent variable**.
  - x is called the ***independent variable***.





What lines "***really***" best fit each case?

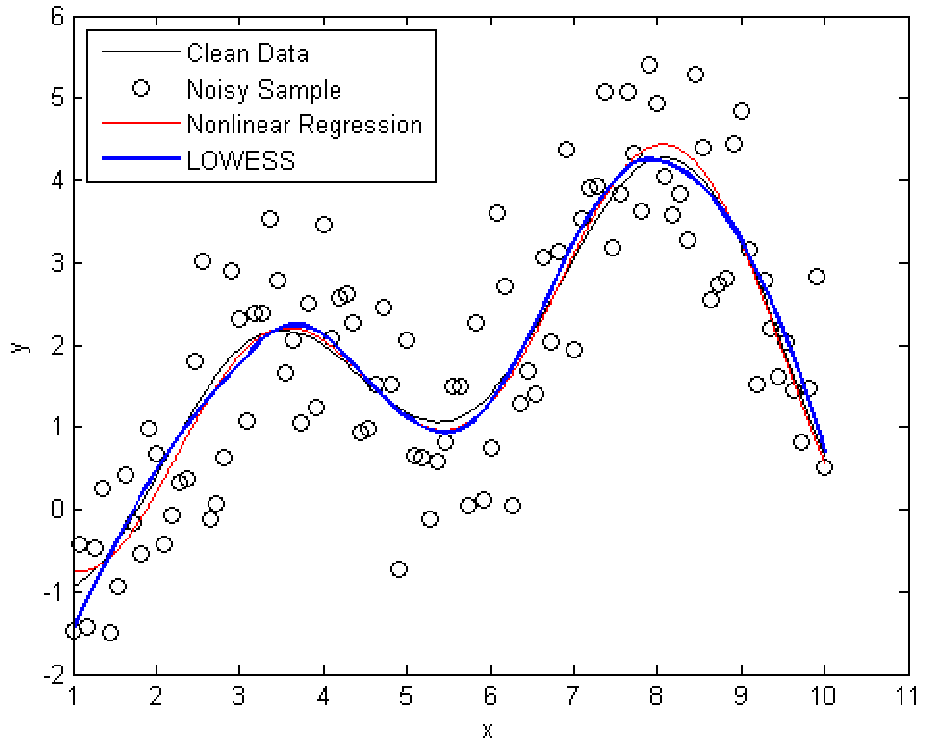


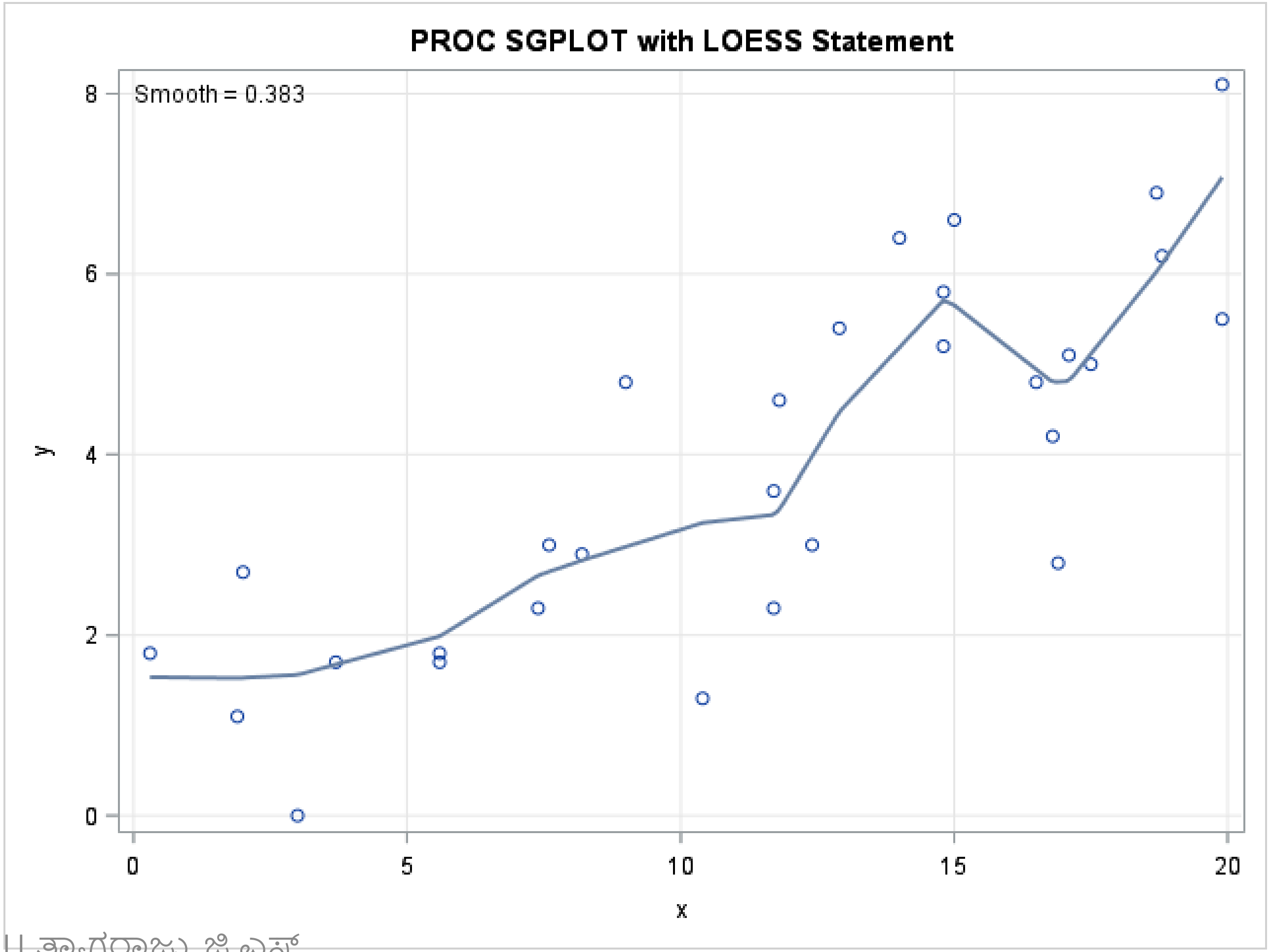
1.2 Locally-weighted Regression

- **Basic idea:** Locally weighted regression constructs an explicit approximation to **f** over a local region surrounding $\mathbf{x_q}$**.**
- **LWR** uses nearby or distance weighted training examples to form this local approximation to f.
- The target function is approximated using ***linear function , quadratic or a multilayer neural network or some other functional form***.

Loess/Lowess Regression

- Loess regression is a nonparametric technique that uses ***local weighted*** regression to fit a **smooth curve** through points in a scatter plot.





Lab Program 10

- **Implement the non-parametric Locally Weighted Regression (LOWESS) algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.**

Lowess Algorithm

- Locally weighted regression is a very powerful non-parametric model used in statistical learning .Given a *dataset* **X**, **y**, we attempt to find a *model* parameter β**(x)** that minimizes *residual sum of **weighted** squared errors*. The weights are given by a *kernel function(k or w)* which can be chosen arbitrarily .

<u>Algorithm</u>

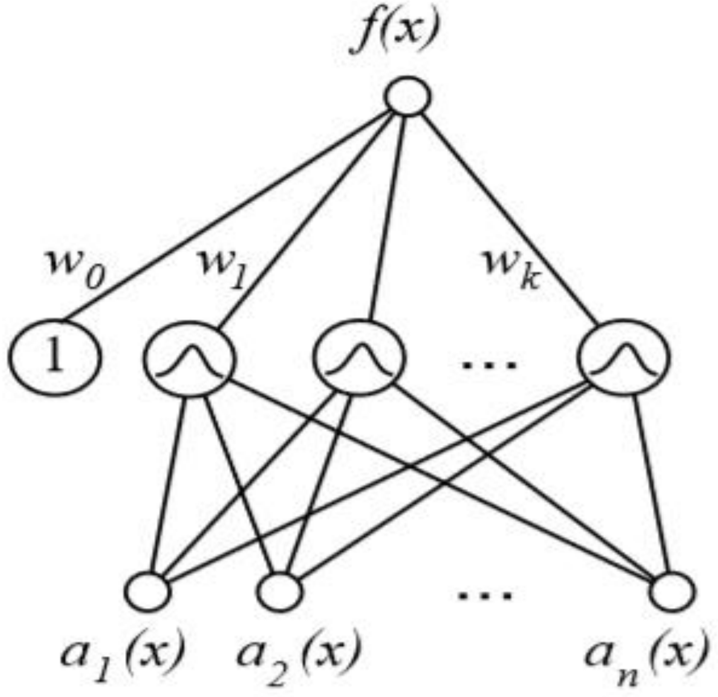
1. **Read the Given data Sample to X and the curve (linear or non linear) to Y**
2. **Set the value for Smoothening parameter or Free parameter say τ**
3. **Set the bias /Point of interest set X0 which is a subset of X**
4. **Determine the weight matrix using :**
$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$
5. **Determine the value of model term parameter β using :** $\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$
6. **Prediction = x_0*β**

Source Code

1.3 Radial basis Function Networks

- One approach to function approximation that is closely related to distance *weighted regression and also to artificial neural network is* ***learning with radial basis funct****ions .*
- ***Global approximation to target function*** *in terms of linear combination of local approximations*
- *Used e.g. for image classification*
- In this approach the learned hypothesis is a function of the form

$$f(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$$



where $a_i(x)$ are the attributes describing instance x, and

$$f(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$$

One common choice for $K_u(d(x_u, x))$ is

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

Training of RBF network

- Given a set of training examples of the target function, RBF networks are typically trained in a **two-stage process.**
- **First, the number k of hidden units is d**etermined and each hidden unit **u** is defined by choosing the values of **x_u and σ_u^$_2$:** that define its kernel function **K_u(d(x_u, x))**.
- **Second, the weights w**, are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

- Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently.

1.4 Case Based Reasoning

- **In CBR, instances are typically represented using symbolic descriptions**, and the methods used to retrieve similar instances are correspondingly more elaborate.
- CBR has been applied to problems such
  - ***as conceptual design of mechanical devices based on a stored library of previous designs***
  - ***reasoning about new legal cases based on previous rulings*** and
  - ***solving planning* and *scheduling problems by reusing and combining portions*** of previous solutions to similar problems

Case Based Reasoning

Case-Based Reasoning is instance-based learning applied to instances with symbolic logic descriptions

```
((user-complaint error53-on-shutdown)
 (cpu-model PowerPC)
 (operating-system Windows)
 (network-connection PCIA)
 (memory 48meg)
 (installed-applications Excel Netscape VirusScan)
 (disk 1gig)
 (likely-cause ???))
```

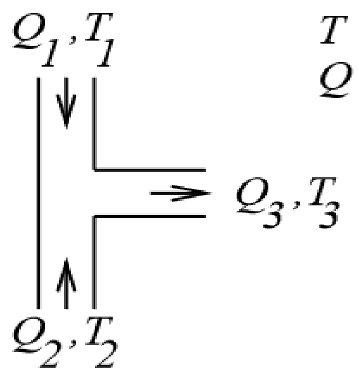
Case Based Reasoning in CADET

- The CADET system (Sycara et al. 1992) employs case- based reasoning to assist in the conceptual design of ***simple mechanical devices such as water faucets.***
- It uses a library containing approximately ***75 previous designs and design fragments*** to suggest conceptual designs to meet the ***specifications of new design problems.***
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

Case Based Reasoning in CADET

A stored case: T−junction pipe

Structure:



T = temperature
Q = waterflow

Function:

$$\begin{tikzcd}
Q_1 \arrow[dr, "+"] & \\
& Q_3 \\
Q_2 \arrow[ur, "+"'] & \\
T_1 \arrow[dr, "+"] & \\
& T_3 \\
T_2 \arrow[ur, "+"'] &
\end{tikzcd}$$

A problem specification: Water faucet

Structure:

?

Function:

$$\begin{tikzcd}
C_t \arrow[r, "+"] \arrow[ddr, "+"] & Q_c \arrow[dr, "+"] \arrow[ddddr, "-"] & \\
& & Q_m \\
C_f \arrow[uur, "+"] \arrow[r, "+"'] & Q_h \arrow[ur, "+"] \arrow[ddr, "+"] & \\
& T_c \arrow[dr, "+"] & \\
& & T_m \\
& T_h \arrow[ur, "+"'] &
\end{tikzcd}$$

- **Qc,** refers to the flow of cold water into the faucet,
- **Qh** to the input flow of hot water, and
- **Qm**, to the single mixed flow out of the faucet.
- Similarly, Tc, Th, and Tm , refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable Ct, denotes the control signal for temperature that is input to the faucet, and
- **Cf** denotes the control signal for waterflow.

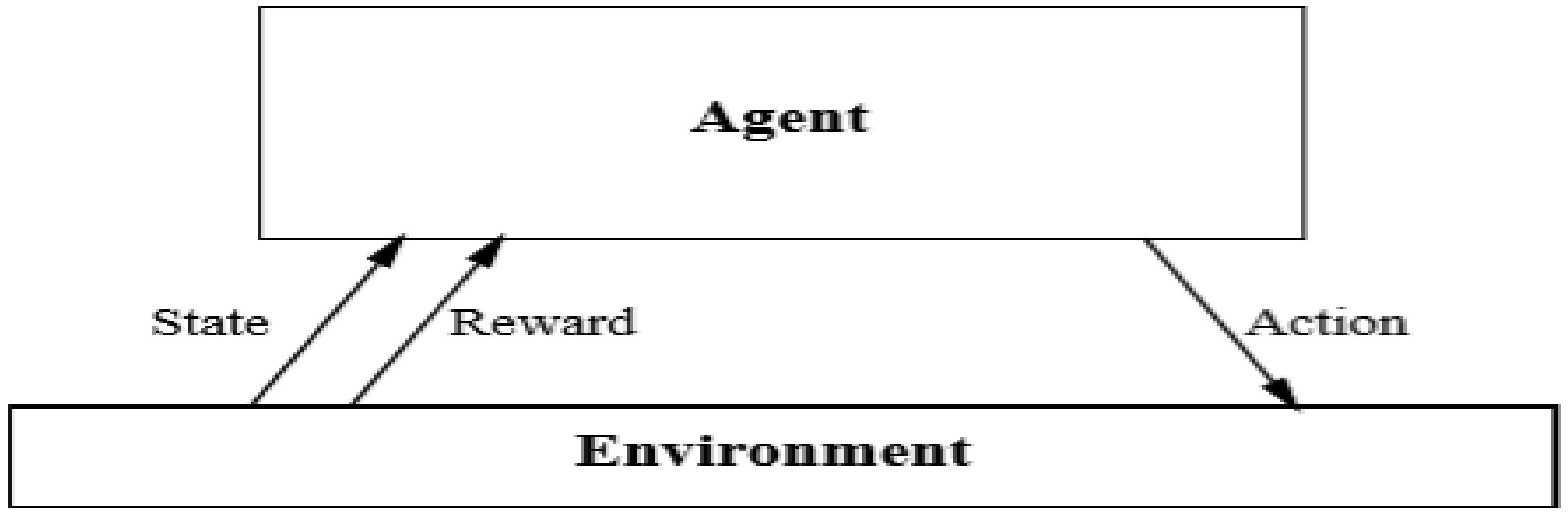
2. Reinforced Learning

- Reinforcement learning addresses the question of how an **autonomous agent that senses and acts in its environment** can learn to choose optimal actions to achieve its goals.
- This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games***. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state.***
    - *For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to* ***learn from this indirect, delayed reward****, to choose sequences of actions that produce the greatest cumulative reward.*

Building a Learning Robot

- Consider building a learning robot. The robot, or agent, has a *set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.*
    - For example, a mobile robot may have sensors such as a ***camera and sonars***, and actions such as "**move forward**" and "**turn**.“
- Its task is to learn a control strategy, or policy, for choosing actions that achieve its goals.
    - For example, the robot may have a goal of ***docking onto its battery charger whenever its battery level is low.***

2.Reinforcement Learning



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \quad , \text{ where } 0 \leq \gamma < 1$$

Explanation

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- It can perform any of a set of possible actions **A**. Each time it performs an action **a**, in some state $\mathbf{s}_t$ the agent receives a real-valued reward **r**, that indicates the immediate value of this state-action transition. This produces a sequence of states $\mathbf{s}_i$, actions $\mathbf{a}_i$**,** and immediate rewards $\mathbf{r}_i$ as shown in the figure.
- The agent's task is to learn a control policy, **π : S -> A,** that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay

The aspects which makes RL different from other

The reinforcement learning problem differs from other function approximation tasks in several important respects

- **Delayed reward:** ***Determining which of the actions in its sequence are to be credited with producing the eventual rewards***
- **Exploration :** The learner faces a tradeoff in choosing whether to *favor exploration of unknown states and actions* (**to gather new information**), or *exploitation of states and actions that it has already learned will yield high reward* (**to maximize its cumulative reward**).
- **Partially observable states:** in many practical situations sensors provide only partial information.
- **Life-long learning**

Agents Learning Task (Value Function)

- The task of the agent is to learn a policy, **π : S -> A**, for selecting its next action **a**, based on the current observed state s_t; that is, **π(s_t) = a,**
- How shall we specify precisely which policy **π** we would like the agent to learn?
- One obvious approach is to require the policy that produces the greatest ***possible cumulative reward for the robot over time***.
- To state this requirement more precisely, we define the **cumulative value** V$^{\pi}$ (s_t) achieved by following an arbitrary policy **n** from an arbitrary initial state s_t as follows:

$$
\begin{aligned}
V^{\pi}(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\
&\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\end{aligned}
$$

Value Function

To begin, consider deterministic worlds...

For each possible policy π the agent might adopt, we can define an evaluation function over states

$$\begin{aligned} V^{\pi}(s) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

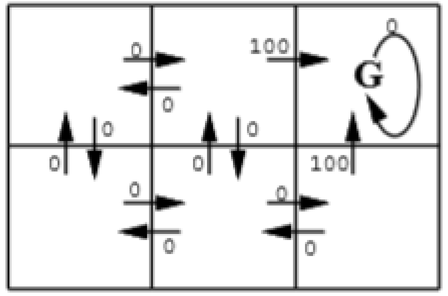
where $r_t, r_{t+1}, \ldots$ are generated by following policy π starting at state s

Restated, the task is to learn the optimal policy π^*

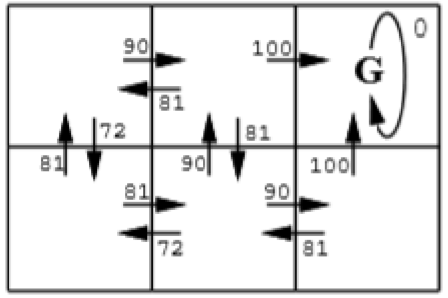
$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s)$$

The quantity V^{π} (s_t) is often called the *discounted cumulative reward achieved by policy* **π.**

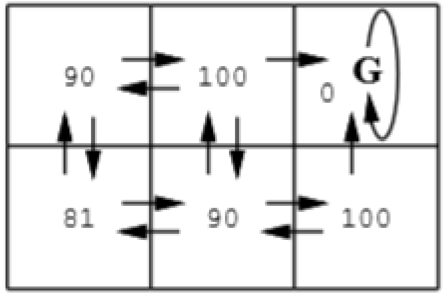
- The six grid squares in this diagram represent ***six possible states, or locations, for the agent***.
- Each arrow in the diagram represents a ***possible action the agent can take to move from one state to another***.
- The number associated *with each arrow represents the immediate reward* ***r(s,a)*** *the agent receives* if it executes the corresponding state-action transition.
- Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into **the state labeled G.**
- It is convenient to think ***of the state G as the goal state,*** because the only way the agent can receive reward, in this case, ***is by entering this state.***
- Note in this particular environment, the only action available to the agent once it enters the ***state G is to remain in this state***.
- For this reason, we call **G** an absorbing state.



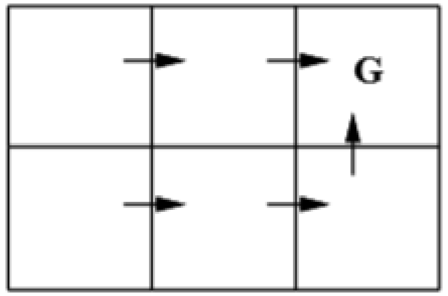
$r(s,a)$ (immediate reward) values



$Q(s,a)$ values

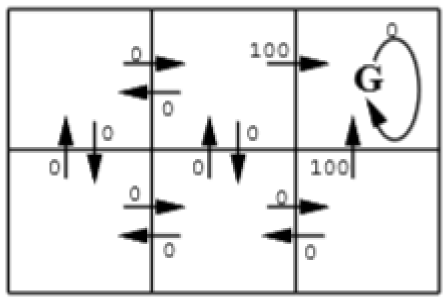


$V^*(s)$ values

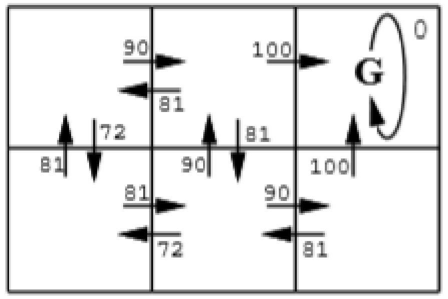


One optimal policy

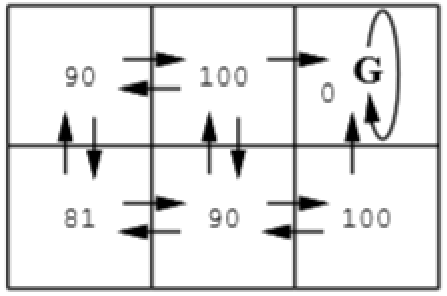
- A simple deterministic world to illustrate the basic concepts of ***Q-learning***. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, **r(s,a)** gives reward 100 for actions entering the goal state G, and zero otherwise.
- Values of $V^{\pi}(s)$and Q(s, a) follow from r(s, a), and the discount factor Y=0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.



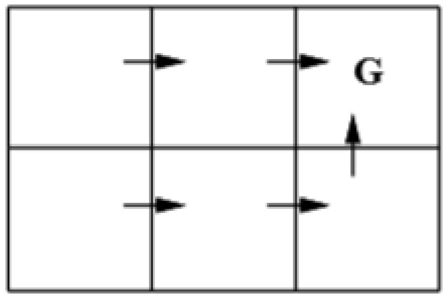
$r(s,a)$ (immediate reward) values



$Q(s,a)$ values

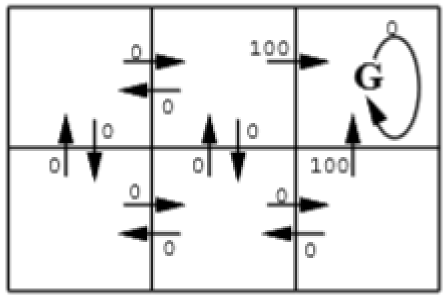


$V^*(s)$ values

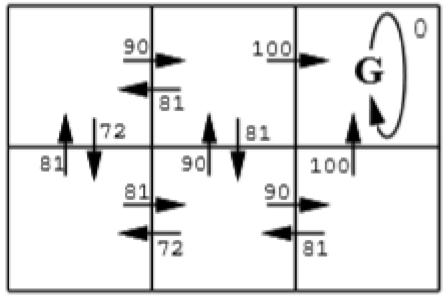


One optimal policy

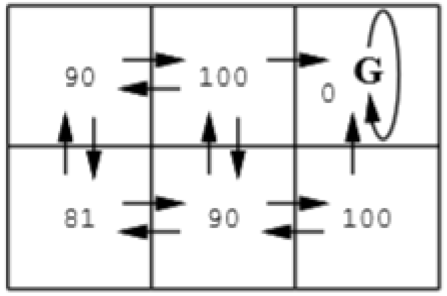
- The diagram at the right of Figure shows the values of V* for each state.
- For example, consider the bottom right state in this diagram. The value of V* for this state is 100 because the optimal policy in this state selects the "*move up*" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of V* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is
- $0 + Y\ 100 + Y^{2}0 + Y^{3}\ 0 + ... = 90$



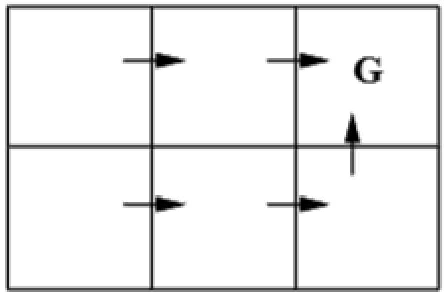
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^{*}(s)$ values



One optimal policy

What to Learn

We might try to have agent learn the evaluation function V^{π^*} (which we write as V^*)

It could then do a lookahead search to choose best action from any state s because

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

A problem:

- This works well if agent knows $\delta : S \times A \rightarrow S$, and $r : S \times A \rightarrow \Re$
- But when it doesn't, it can't choose actions this way

Q Function

Define new function very similar to V^*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

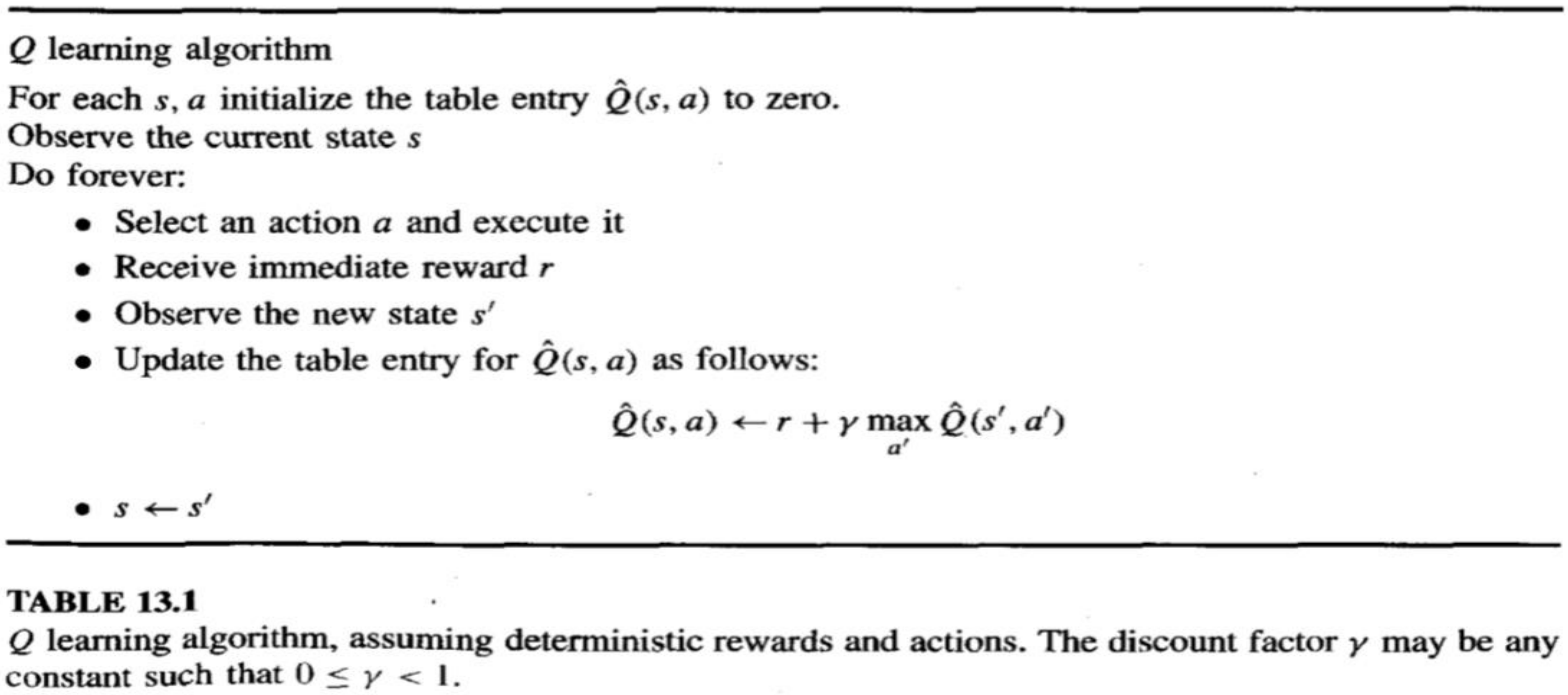
If agent learns Q, it can choose optimal action even without knowing δ!

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

Q is the evaluation function the agent will learn

An Algorithm for Learning Q



Q learning algorithm

```
For each $s, a$ initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state $s$
Do forever:
    • Select an action $a$ and execute it
    • Receive immediate reward $r$
    • Observe the new state $s'$
    • Update the table entry for $\hat{Q}(s, a)$ as follows:
          $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
    • $s \leftarrow s'$
```

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

Training Rule to Learn *Q*

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write Q recursively as

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Nice! Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

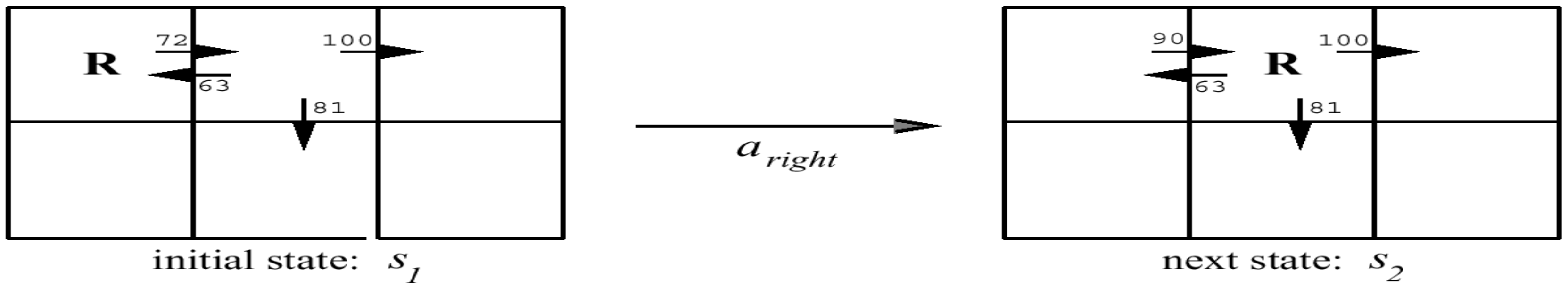
$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s

Q Learning for Deterministic Worlds

```
For each $s, a$ initialize table entry $\hat{Q}(s, a) \leftarrow 0$
Observe current state $s$
Do forever:
    • Select an action $a$ and execute it
    • Receive immediate reward $r$
    • Observe the new state $s'$
    • Update the table entry for $\hat{Q}(s, a)$ as follows:
          $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
    • $s \leftarrow s'$
```

Updating $\hat{Q}$



$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

notice if rewards non-negative, then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a)$$

and

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$$

$\hat{Q}$ converges to Q. Consider case of deterministic world where see each $\langle s, a \rangle$ visited infinitely often.

Proof: Define a full interval to be an interval during which each $\langle s, a \rangle$ is visited. During each full interval the largest error in $\hat{Q}$ table is reduced by factor of γ

Let $\hat{Q}_n$ be table after n updates, and Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$$

For any table entry $\hat{Q}_n(s,a)$ updated on iteration $n+1$, the error in the revised estimate $\hat{Q}_{n+1}(s,a)$ is

$$\begin{aligned} |\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) \\ &\quad -(r + \gamma \max_{a'} Q(s',a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')| \\ &\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')| \\ |\hat{Q}_{n+1}(s,a) - Q(s,a)| &\leq \gamma \Delta_n \end{aligned}$$

Nondeterministic Case(Cont')

Q learning generalizes to nondeterministic worlds

Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a)+\alpha_n[r+\max_{a'} \hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

Can still prove convergence of $\hat{Q}$ to Q [Watkins and Dayan, 1992]

Temporal Difference Learning

Q learning: reduce discrepancy between successive Q estimates

One step time difference:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Why not two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Or n?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) \right.$$

Temporal Difference Learning(Cont')

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda) \Big[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)$$

Equivalent expression:

$$\begin{aligned} Q^{\lambda}(s_t, a_t) \; = r_t + \gamma [\; & (1-\lambda) \max_a \hat{Q}(s_t, a_t) \\ & + \lambda \; Q^{\lambda}(s_{t+1}, a_{t+1})] \end{aligned}$$

TD(λ) algorithm uses above training rule

- Sometimes converges faster than Q learning
- converges for learning V^* for any $0 \leq \lambda \leq 1$ (Dayan, 1992)
- Tesauro's TD-Gammon uses this algorithm

Subtleties and Ongoing Research

- Replace $\hat{Q}$ table with neural net or other generalizer
- Handle case where state only partially observable
- Design optimal exploration strategies
- Extend to continuous action, state
- Learn and use $\hat{\delta} : S \times A \rightarrow S$
- Relationship to dynamic programming

Q-table initialised at zero

	UP	DOWN	LEFT	RIGHT
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

After few episodes

	UP	DOWN	LEFT	RIGHT
0	0	0	0	0
1	0	0	0	0
2	0	2.25	2.25	0
3	0	0	5	0
4	0	0	0	0
5	0	0	0	0
6	0	5	0	0
7	0	0	2.25	0
8	0	0	0	0

Eventually

	UP	DOWN	LEFT	RIGHT
0	0	0	0.45	0
1	0	1.01	0	0
2	0	2.25	2.25	0
3	0	0	5	0
4	0	0	0	0
5	0	0	0	0
6	0	5	0	0
7	0	0	2.25	0
8	0	0	0	0