ARTIFICIAL INTELLIGENCE

UNIT – III

Syllabus

Reinforcement Learning: Introduction, Passive Reinforcement Learning, Active Reinforcement Learning, Generalization in Reinforcement Learning, Policy Search, applications of RL .

Natural Language Processing: Language Models, Text Classification, Information Retrieval, Information Extraction.

Chapter – 1

Reinforcement Learning

3.1.1. INTRODUCTION:

- A supervised learning agent needs to be told the correct move for each position it encounters, but such feedback is seldom available.
- In the absence of feedback from a teacher, an agent can learn a transition model for its own moves and can perhaps learn to predict the opponent's moves, but *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.*
- The agent needs to know that something good has happened when it (accidentally) checkmates the opponent, and that something bad has happened when it is checkmated—or vice versa, if the game is suicide chess.

  This kind of feedback is called a reward, or reinforcement.
- In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently.
- In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement.
- In animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards.
- Rewards, where they served to define optimal policies in Markov decision processes (MDPs).
- An optimal policy is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.
- Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.
- In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels.

  Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the concept manageable, we will concentrate on simple environments and simple agent designs.
- Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs first :

→ A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.

→ A **Q-learning** agent learns an **action-utility function**, or **Q-function**, giving the ex-pected utility of taking a given action in a given state.

A **reflex agent** learns a policy that maps directly from states to actions.

- A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead.
- A Q-learning agent, on the other hand, can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment.
- On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn

(i) Passive learning, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning a model of the environment.

(ii) Active learning, where the agent must also learn what to do.

→ The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it.

3.1.2. PASSIVE REINFORCEMENT LEARNING :

- To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment.
- In passive learning, the agent's policy π is fixed: in state s, it always executes the action π(s).

Its goal is simply to learn how good the policy is—that is, to learn the utility function U π(s).

- We will use as our example the 4 × 3 world shows a policy for that world and the corresponding utilities.
- Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm.

The main difference is that the passive learning agent does not know the **transition model** P(s | s, a), which specifies the probability of reaching state s from state s after doing action a; nor does it now the **reward function** R(s), which specifies the reward for each state.

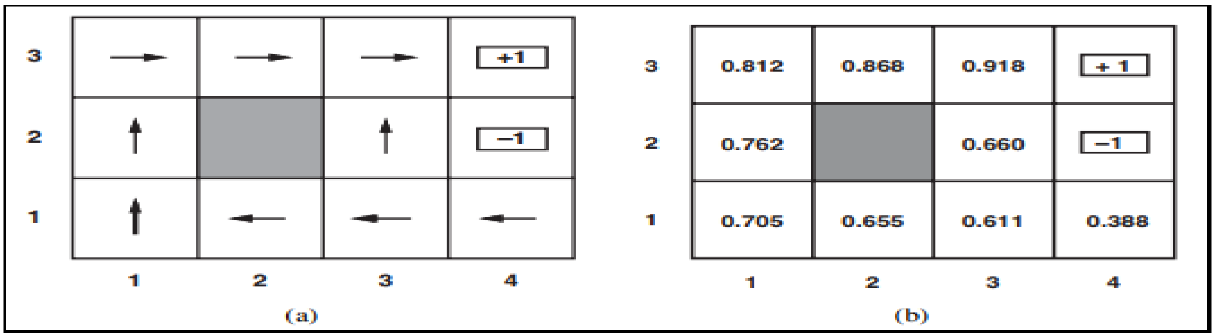


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

- The agent executes a set of trials in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state.
- The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed. As in Equation we write :

where R(s) is the reward for a state, St (a random variable) is the state reached at time t when executing policy π, and S0 = s. We will include a discount factor γ in all of our equations, but for the 4 × 3 world we will set γ = 1.

(i) Direct utility estimation :

- A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960).
- The idea is that the utility of a state is the expected total reward from that state onward (called the expected **reward-to-go**), and each trial provides a *sample* of this quantity for each state visited.
- For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1) and so on.

Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table.

- It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output.
- Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known.
- Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! *The utility of each state equals its own reward plus the expected utility of its successor states.*

That is, the utility values obey the Bellman equations for a fixed policy :

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$$

(ii) Adaptive dynamic programming :

- An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method.

For a passive learning agent, this means plugging the learned transition model P(s |s, π(s)) and the observed rewards R(s) into the Bellman equations to calculate the utilities of the states.

- Alternatively, we can adopt the approach of **modified policy iteration** , using a simplified value iteration process to update the utility estimates after each change to the learned model.

- Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.
- The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state.

In the simplest case, we can represent the transition model as a table of probabilities.

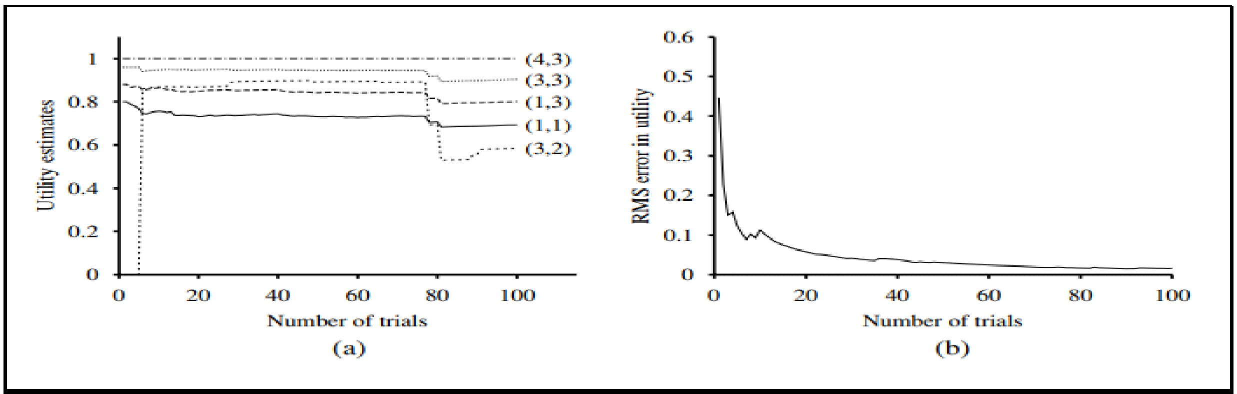


Figure 21.3 The passive ADP learning curves for the 4 × 3 world, given the optimal policy

- There are two mathematical approaches that have this flavor.
- The first approach, **Bayesian reinforcement learning**, assumes a prior probability P (h) for each hypothesis h about what the true model is; the posterior probability P (h | **e**) is obtained in the usual way by Bayes' rule given the observations to date.

The second approach, derived from **robust control theory**, allows for a *set* of possible models H and defines an optimal robust policy as one that gives the best outcome in the *worst case* over H:

(iii) Temporal-difference learning :

- Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem.
- Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations.
- Consider, for example, the transition from (1,3) to (2,3) Suppose that, as a result of the first trial, the utility estimates are U π(1, 3) = 0.84 and U π(2, 3) = 0.92.

Now, if this transition occurred all the time, we would expect the utilities to obey the equation

$$U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$$

- So U π(1, 3) would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s, we apply the following update to U π(s):
- Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference**, or TD, equation.

3.1.3. ACTIVE REINFORCEMENT LEARNING:

- A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take.
- Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.
- The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations , which we repeat here for convenience:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' \mid s, a) U(s') .$$

(i) Exploration :

- In Figure it shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step.
- The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1),(3,2), and (3,3).
- After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3).
- We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.

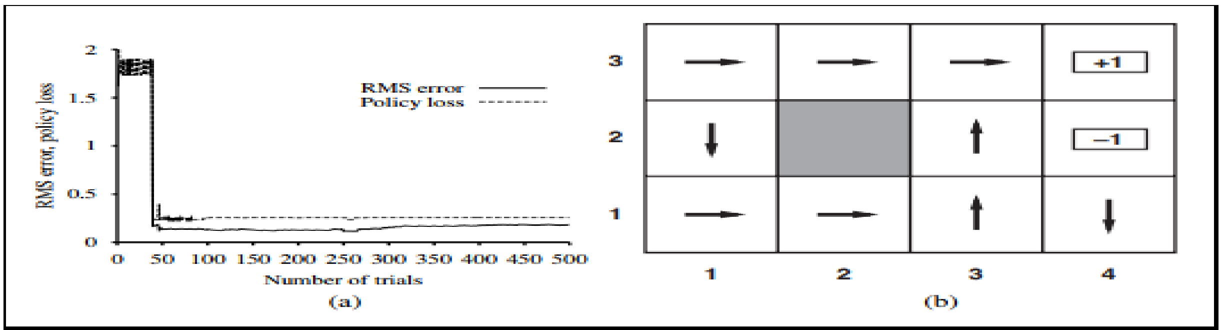


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

- By improving the model, the agent will receive greater rewards in the future. An agent therefore must make a tradeoff between **exploitation** to maximize its reward.
- Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that nowledge into practice. With greater understanding, less exploration is necessary.
- A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.

An ADP agent using such a scheme will eventually learn the true environment model.

- In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any).
- An n-**armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?
- This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter.

  However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

(ii) Learning an action-utility function :

- Now that we have an active ADP agent, let us consider how to construct an active temporal difference learning agent.
- The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead.

  The model acquisition problem for the TD agent is identical to that for the ADP agent.
- Suppose the agent takes a step that normally leads to a good destination, but because of non-determinism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously.
- There is an alternative TD method, called **Q-learning**, which learns an action-utility representation instead of learning utilities. We will use the notation Q(s, a) to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(s,a) \ .$$

- Q-functions may seem like just another way of storing utility information, but they have avery important property:

  "a TD agent that learns a Q-function does not need a model of the
  form P(s | s, a), *either for learning or for action selection."*
- For this reason, Q-learning is called a **model-free** method.
- As with utilities, we can write a constraint equation :

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a') \ .$$

```
function Q-LEARNING-AGENT(percept) returns an action
   inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
   persistent: $Q$, a table of action values indexed by state and action, initially zero
               $N_{sa}$, a table of frequencies for state–action pairs, initially zero
               $s, a, r$, the previous state, action, and reward, initially null

   if TERMINAL?($s$) then $Q[s, None] \leftarrow r'$
   if $s$ is not null then
       increment $N_{sa}[s, a]$
       $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
   $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a']), r'$
   return $a$
```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s,a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

- Q-learning has a close relative called SARSA (for State-Action-Reward-State-Action).
- The update rule for SARSA is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma\, Q(s',a') - Q(s,a))\ ,$$

3.1.4. GENERALIZATION IN REINFORCEMENT LEARNING :

- So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple.
- Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger.
- With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question.
- Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 1020 and 1040 states, respectively.
- It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game.
- One way to handle such problems is to use **function approximation**, which simplymeans using any sort of representation for the Q-function other than a lookup table.
- The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form.
- For example, we described an **evaluation function** for chess that is represented as a weighted linear

function of a set of **features** (or **basis functions**) f1, . . . , fn:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

- A reinforcement learning algorithm can learn values for the parameters θ = θ1, . . . , θn such that the evaluation function Uˆθ approximates the true utility function.
- Instead of, say, values in a table, this function approximator is characterized by, say, n = 20 parameters an *enormous* compression.
- Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers.
- If the approximation is good enough, however, the agent might still play excellent chess.
- Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit.
- *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.*
- That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states.
- For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial.
- Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that Uˆθ(1, 1), currently 0.8, is too large and must be reduced.
- There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.
- Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state.
- For a *partially observable* environment, the learning problem is much more difficult.
- If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters .
- Inventing the hidden variables and learning the model structure are still open problems.

3.1.5. POLICY SEARCH :

- The final approach we will consider for reinforcement learning problems is called policy search.
- In some ways, policy search is the simplest of all the methods, the idea is to keep twiddling the policy as long as its performance improves, then stop.

- Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions.
- We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space .
- For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(s,a)$$

- Each Q-function could be a linear function of the parameters θ, or it could be a nonlinear function such as a neural network.
- Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q functions, then policy search results in a process that learns Q-functions.
- *This process is not the same as Q-learning!* In Q-learning with function approximation, the algorithm finds a value of θ such that Qˆθ is "close" to Q, the optimal Q-function.
- Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially.

One problem with policy representations of the kind is that the policy is a *discontinuous* function of the parameters when the actions are discrete.

- That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another.
- This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult.

For this reason, policy search methods often use a **stochastic policy** representation πθ(s, a), which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax function**:

$$\pi_\theta(s,a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}$$

- For the case of a stochastic policy πθ(s, a), it is possible to obtain an unbiased estimate of the gradient at θ, θρ(θ), directly from the results of trials executed at θ.
- In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0,a)R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0,a))R(a)$$

Suppose that we have N trials in all and the action taken on the jth trial is aj. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0,a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0,a))R(a)}{\pi_\theta(s_0,a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0,a_j))R(a_j)}{\pi_\theta(s_0,a_j)}$$

- For the sequential case, this generalizes to :

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j)) R_j(s)}{\pi_\theta(s, a_j)}$$

- Policy search is carried out by evaluating each candidate policy using the *same* set of random sequences to determine the action outcomes.

It can be shown that the number of random sequences required to ensure that the value of *every* policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

3.1.6. APPLICATIONS OF REINFORCEMENT LEARNING:

(i) Applications to game playing :

→ The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers program written by Arthur Samuel (1959, 1967).

- Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation to update the weights.
- There were some significant differences, however, between his program and current methods.
- First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree.
- Second difference was that the program did *not* use any observed rewards .

→ **Gerry Tesauro's backgammon program TD-GAMMON** (1992) forcefully illustrates the potential of reinforcement learning techniques.

- In earlier work Tesauro tried learning a neural network representation of Q(s, a) directly from examples of moves labeled with relative values by a human expert.
- This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts.
- The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game.

(ii) Application to robot control :

- The setup for the famous cart–pole balancing problem, also known as the **inverted pendulum.**

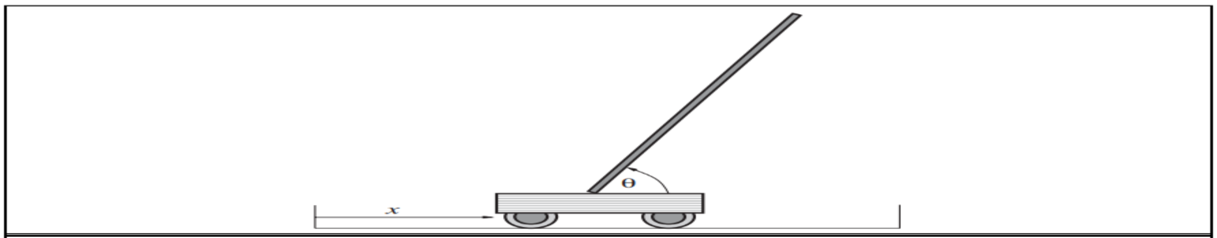


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

- The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown.
- The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $x^{\cdot}$, and $\theta^{\cdot}$ are continuous.
- The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

  The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials.
- Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation.
- The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track.
- Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence.
- It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect.
- Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network.
- Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

➔ Still more impressive is the application of reinforcement learning to **helicopter flight.**

Figure 21.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy-search algorithm. A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

- This work has generally used policy search as well as the PEGASUS algorithm with simulation based on a learned transition model.

ARTIFICIAL INTELLIGENCE

UNIT – III

<u>Chapter – 2</u>

Natural Language Processing

3.2.1. LANGUAGE MODELS :

- Formal languages, such as the programming languages Java or Python, have precisely defined language models.
- A **language** can be defined as a set of strings; “print(2 + 2)” is a legal program in the language Python, whereas “2)+(2 print” is not.
- Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by set of rules called a **grammar**.
- Formal languages also have rules that define the meaning or **semantics** of a program; for example, the rules say that the “meaning” of “2 + 2” is 4, and the meaning of “1/0” is that an error is signaled.
- Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences.
- Everyone agrees that “Not to be invited is sad” is a sentence of English, but people disagree on the grammaticality of “To be not invited is sad.”
- Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set.

  P(S = words)
- Natural languages are also **ambiguous**. Because we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.
- Finally, natural languages are difficult to deal with because they are very large, and constantly changing.
- Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

(i) ***N*-gram character models** :

- Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages).
- Thus, one of the simplest language models is a probability distribution over sequences of characters.
- We write P(C1:N) for the probability of a sequence of N characters, C1 through CN.
- In one Web collection, P(“the”) = 0.027 and P(“zgq”) = 0.000000002.
- A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case “unigram” for 1-gram, “bigram” for 2-gram, and “trigram” for 3-gram.
- A model of the probability distribution of n-letter sequences is thus called an n**-gram model**. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.)
- An n-gram model is defined as a **Markov chain** of order n - 1.
- In a Markov chain the probability of character ci depends only on the immediately preceding characters, not on any other characters.
- So in a trigram model (Markov chain of order 2) we have :

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1})$$

- We can define the probability of a sequence of characters P(c1:N) under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1})$$

- We call a body of text a **corpus** (plural *corpora*), from the Latin word for *body*.
- What can we do with n-gram character models? One task for which they are well suited is **language identification** .
- For Example, given a text, determine what natural language it is written in.
- This is a relatively easy task; even with short texts such as "Hello, world" or "Wie geht es dir," it is easy to identify the first as English and the second as German.
- Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.
- One approach to language identification is to first build a trigram character model of each candidate language, where the variable L ranges over languages.
- That gives us a model of **P**(Text | Language), but we want to select the most probable language given the text, so we apply Bayes' rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned} \ell^* &= \underset{\ell}{\operatorname{argmax}}\ P(\ell \mid c_{1:N}) \\ &= \underset{\ell}{\operatorname{argmax}}\ P(\ell) P(c_{1:N} \mid \ell) \\ &= \underset{\ell}{\operatorname{argmax}}\ P(\ell) \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}, \ell) \end{aligned}$$

(ii) Smoothing *n*-gram models :

- The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution.
- For common character sequences such as " _th" any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, " _ht" is very uncommon—no dictionary words start with ht.

  The process of adjusting the probability of low-frequency counts is called **smoothing**.
- The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for P (X = true) should be 1/(n+2).
- That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly.
- A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n - 1)-grams.
- **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as :

$$\widehat{P}(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i)$$

- where λ3 + λ2 + λ1 = 1. The parameter values λi can be fixed, or they can be trained with an expectation–maximization algorithm.
- It is also possible to have the values of λi depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models.

(iii) Model evaluation :

- With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation.
- Split the corpus into a training corpus and a validation corpus.Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.
- The evaluation can be a task-specific metric, such as measuring accuracy on language identification.
- Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better.
- This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue.
- A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as :

$$Perplexity(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}$$

- Perplexity can be thought of as the reciprocal of probability, normalized by sequence length.
- It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100.
- If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

(iv) N-gram word models :

- Now we turn to n-gram models over words rather than characters.
- All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**—the set of symbols that make up the corpus and the model—is larger.
- There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating “A” and “a” as the same symbol or by treating all punctuation as the same symbol.
- But with word models we have at least tens of thousands of symbols, and sometimes millions.
- The wide range is because it is not clear what constitutes a word.
- In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in.

- Word n-gram models need to deal with **out of vocabulary** words.
- With character models, we didn't have to worry about someone inventing a new letter of the alphabet.
- But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model.

3.2.2. TEXT CLASSIFICATION :

- We now consider in depth the task of **text classification**, also known as **categorization**: givena text of some kind, decide which of a predefined set of classes it belongs to.
- Language identification and genre classification are examples of text classification
- **spam detection** classifying an email message as spam or not-spam(ham).
- A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox.
- Note that we have two complementary ways of talking about classification.
- In the language-modeling approach, we define one n-gram language model for **P**(Message | spam) by training on the spam folder, and one model for **P**(Message | ham) by training on the inbox.
- Then we can classify a new message with an application of Bayes' rule:

$$\underset{c\in\{spam,ham\}}{\operatorname{argmax}} P(c \mid message) = \underset{c\in\{spam,ham\}}{\operatorname{argmax}} P(message \mid c)\, P(c)$$

- where P (c) is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.
- If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero.
- This unigram representation has been called the **bag of words** model.
- You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time.
- The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text.
- Higher-order n-gram models maintain some local notion of word order.
- It can be expensive to run algorithms on a very large feature vector, so often a process of **feature selection** is used to keep only the features that best discriminate between spam and ham.
- Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression.
- All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

(i) **Classification by data compression** :

- Another way to think about classification is as a problem in **data compression**.
- A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original.
- For example, the text "0.142857142857142857" might be compressed to "0.[142857]*3."
- To do classification by compression, we first lump together all the spam training messages and compress them as a unit.
- We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result.
- We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class.

<u>3.2.3. INFORMATION RETRIEVAL</u> :

- Information retrieval is the task of finding documents that are relevant to a user's need for information.
- The best-known examples of information retrieval systems are search engines on the World Wide Web.
- A Web user can type a query such as "AI book" into a search engine and see a list of relevant pages.

- An **information retrieval** (henceforth **IR**) system can be **characterized** by :

→**A corpus of documents.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.

➔ **Queries posed in a query language**. A query specifies what the user wants to know.The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book].

➔ →**A result set.** This is the subset of documents that the IR system judges to be **relevant** to the query.

→ **A presentation of the result set.** This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space, rendered as a two-dimensional display.

- The earliest IR systems worked on a **Boolean keyword model**. Each word in the documentcollection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not.

  The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true.

- This model has the advantage of being simple to explain and implement.
- However, it has some **disadvantages:**

- ➔ First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation.
- ➔ Second, Boolean expressions are unfamiliar to users who are not programmers or logicians.
- ➔ Third, it can be hard to formulate an appropriate query, even for a skilled user.

(i) **IR scoring functions** :

- Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the **BM25 scoring function.**
- A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores.

  In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query.
- **Three factors** affect the weight of a query term:

→ First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query documents that mention "farming" frequently will have higher scores.

→ Second, the inverse document frequency of the term, or IDF. The word "in" appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query.

➔ Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.

- The BM25 function takes all three of these into account.
- Then, given a document dj and a query consisting of the words q1:N, we have :

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k+1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})}$$

- IDF(qi) is the inverse document frequency of word qi, given by :

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5} .$$

(ii) IR system evaluation :

- How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments.
- Traditionally, there have been two measures used in the scoring:

  → recall

  → precision.
- **Precision** measures the proportion of documents in the result set that are actually relevant.

- In our example, the precision is 30/(30 + 10) = .75. The false positive rate is 1 - .75 = .25.
- **Recall** measures the proportion of all the relevant documents in the collection that are in the result set.
- In our example, recall is 30/(30 + 20) = .60. The false negative rate is 1 - .60 = .40.
- In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance.
- All we can do is either estimate recall by sampling or ignore recall completely and just judge precision.

(iii) IR refinements :

- There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.
- One common refinement is a better model of the effect of document length on relevance.
- Singhal *et al.* (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough.
- They propose a *pivoted* document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.
- The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated.
- Many IR systems attempt to account for these correlations.
- The next step is to recognize **synonyms**, such as “sofa” for “couch.” As with stemming, this has the potential for small gains in recall, but can hurt precision.
- As a final refinement, IR can be improved by considering m**etadata**—data outside of the text of the document. Examples include human-supplied keywords and publication data.
- On the Web, hypertext **links** between documents are a crucial source of information.

(iv) The PageRank algorithm :

- **PageRank** was one of the two original ideas that set Google’s search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor text—the underlined text in a hyperlink).
- PageRank was invented to solve the problem of the tyranny of TF scores: if the query is [IBM], how do we make sure that IBM’s home page, ibm.com, is the first result, even if another page mentions the term “IBM” more frequently?

  The idea is that ibm.com has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page.
- But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page.

- Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily.
- What is a highquality site? One that is linked to by other high-quality sites.
- The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page p is defined as:

$$PR(p) = \frac{1-d}{N} + d \sum_i \frac{PR(in_i)}{C(in_i)}$$

- where P R(p) is the PageRank of page p, N is the total number of pages in the corpus, ini are the pages that link in to p, and C(ini) is the count of the total number of out-links on page ini.
- The constant d is a damping factor. It can be understood through the **random surfer model** : imagine a Web surfer who starts at some random page and begins exploring.

(v) The HITS algorithm :

- The Hyperlink-Induced Topic Search algorithm, also known as "Hubs and Authorities" or HITS, is another influential link-analysis algorithm .
- HITS differs from PageRank in several ways.
- First, it is a query-dependent measure: it rates pages with respect to a query.
- Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages
- Both PageRank and HITS played important roles in developing our understanding of Web information retrieval.
- These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

(vi) Question answering :

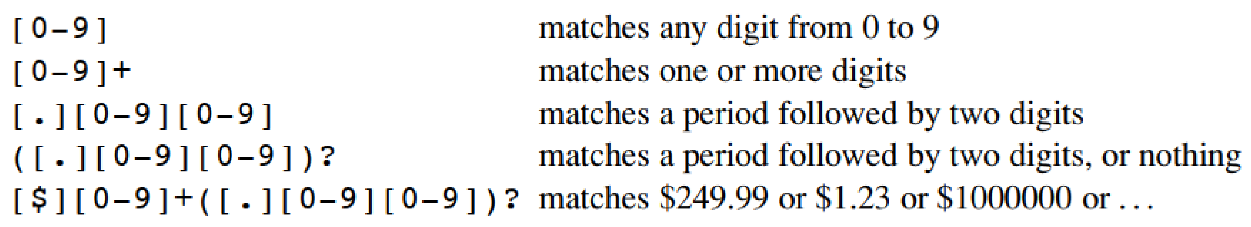
- Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept.
- **Question answering** is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase.
- There have been question-answering NLP (natural language processing) systems since the 1960s, but only since 2001 have such systems used Web information retrieval to radically increase their breadth of coverage.

3.2.4. INFORMATION EXTRACTION :

- Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects.
- A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation.
- In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary.

(i) Finite-state automata for information extraction:

- The simplest type of information extraction system is an **attribute-based extraction** system that assumes that the entire text refers to a single object and the task is to extract attributes of that object.
- For example, the problem of extracting from the text "IBM ThinkBook 970. Our price: $399.00" the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=$399.00}.
- We can address this problem by defining a **template** (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the **regular expression**, or regex.
- Here we show how to build up a regular expression template for prices in dollars:



`[0-9]`	matches any digit from 0 to 9
`[0-9]+`	matches one or more digits
`[.][0-9][0-9]`	matches a period followed by two digits
`([.][0-9][0-9])?`	matches a period followed by two digits, or nothing
`[$][0-9]+([.][0-9][0-9])?`	matches $249.99 or $1.23 or $1000000 or . . .

- Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex.
- For prices, the target regex is as above, the prefix would look for strings such as "price:" and the postfix could be empty.
- The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.
- One step up from attribute-based extraction systems are **relational extraction** systems, which deal with multiple objects and the relations among them.
- Thus, when these systems see the text "$249.99," they need to determine not just that it is a price, but also which object has that price.
- A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions.
- A relational extraction system can be built as a series of **cascaded finite-state transducers**.
- That is, the system consists of a series of small, efficient finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton.

- **FASTUS consists of five stages:**

  1. Tokenization
  2. Complex-word handling
  3. Basic-group handling
  4. Complex-phrase handling
  5. Structure merging

1. FASTUS's first stage is **tokenization**, which segments the stream of characters into tokens (words, numbers, and punctuation). Some tokenizers also deal with markup languages such as HTML, SGML, and XML.
2. The second stage handles **complex words**, including collocations such as "set up" and "joint venture," as well as proper names such as "Bridgestone Sports Co."
3. The third stage handles **basic groups**, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages.
4. The fourth stage combines the basic groups into **complex phrases**.
5. The final stage **merges structures** that were built up in the previous step.

(ii) Probabilistic models for information extraction:

- When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly.
- It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model.
- The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.
- HMM models a progression through a sequence of hidden states, **x**t, with an observation **e**t at each step.
- To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We'll do the second.
- HMMs have **two big advantages** over FSAs for extraction.

  → First, HMMs are probabilistic, and thus tolerant to noise.

  →Second, HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

(iii) Conditional random fields for information extraction :

- One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don't really need.
- Modeling this directly gives us some freedom. We don't need the independence assumptions of the Markov model—we can have an **x**t that is dependent on **x**1.
- A framework for this type of model is the **conditional random field**, or CRF, which models a

conditional probability distribution of a set of target variables given a set of observed variables.

- Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables.
- One common structure is the **linear-chain conditional random field** for representing Markov dependencies among variables in a temporal sequence.
- Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression.

(iv) Ontology extraction from large corpora :

- So far we have thought of information extraction as finding a specific set of relations (e.g.,speaker, time, location) in a specific text (e.g., a talk announcement).
- A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus.
- This is different in three ways:
- First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain.
- Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web .
- Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

(v) Automated template construction :

- Fortunately, it is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be learned, and so on.
- In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:

(“Isaac Asimov”, “The Robots of Dawn”)
(“David Brin”, “Startide Rising”)
(“James Gleick”, “Chaos—Making a New Science”)
(“Charles Dickens”, “Great Expectations”)
(“William Shakespeare”, “The Comedy of Errors”)

- Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles.
- The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings,

(*Author, Title, Order, Prefix, Middle, Postfix, URL*) ,

- where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters after the match, and *URL* is the Web address where the match was made.

(vi) Machine reading :

- Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started.
- To build a large ontology with many thousands of relations, even that amount of work would be onerous; we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database.
- Such a system would be relation-independent; would work for any relation. In practice, these systems work on *all* relations in parallel, because of the I/O demands of large corpora.
- They behave less like a traditional information extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.