Multi-Agent Systems & Machine Learning

(Important Concepts and Applications)

Margert Cooney

Julian Curran

First Edition, 2012

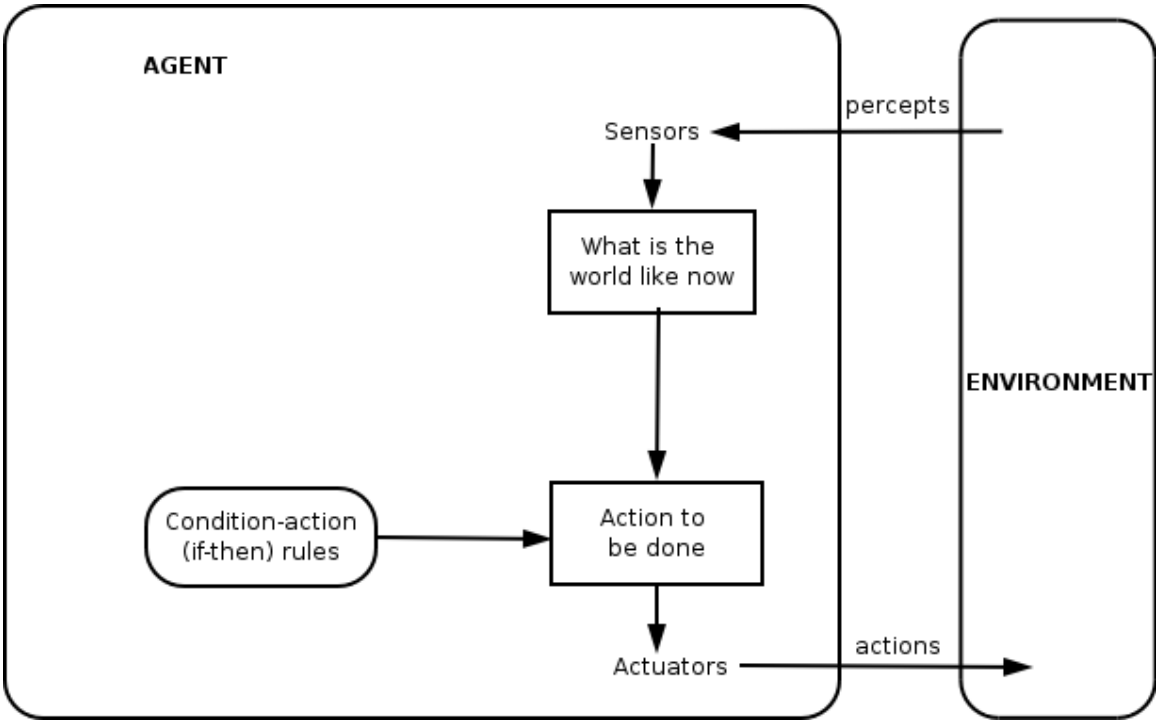
ISBN 978-81-323-0958-1

Published by:
Academic Studio
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

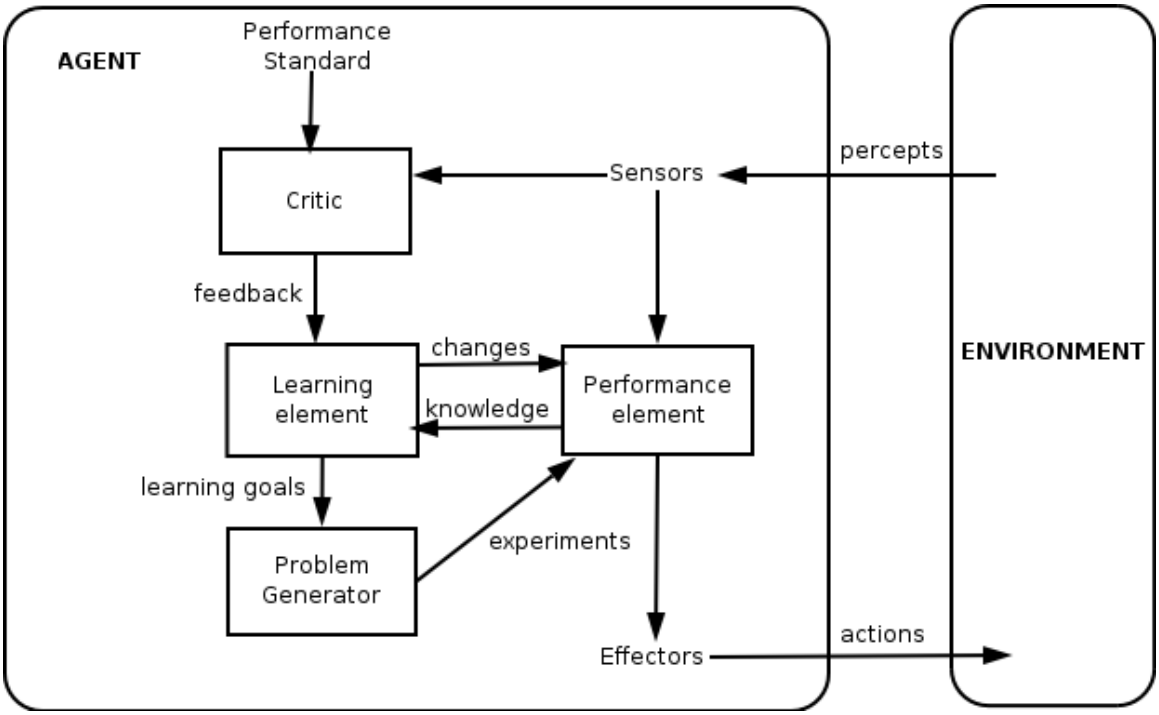
Table of Contents

Chapter 1

Multi-Agent System



Simple reflex agent



Learning agent

A **multi-agent system** (**MAS**) is a system composed of multiple interacting intelligent agents. Multi-agent systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Intelligence may include some methodic, functional, procedural or algorithmic search, find and processing approach.

Topics where multi-agent systems research may deliver an appropriate approach include online trading, disaster response, and modelling social structures.

Overview

The agents in a multi-agent system have several important characteristics:

- **Autonomy**: the agents are at least partially autonomous
- **Local views**: no agent has a full global view of the system, or the system is too complex for an agent to make practical use of such knowledge
- **Decentralization**: there is no designated controlling agent (or the system is effectively reduced to a monolithic system)

Typically multi-agent systems research refers to software agents. However, the agents in a multi-agent system could equally well be robots, humans or human teams. A multi-agent system may contain combined human-agent teams.

Self organisation and self steering

Multi-agent systems can manifest self-organization as well as self-steering and other control paradigms and related complex behaviors even when the individual strategies of all their agents are simple.

When agents can share knowledge using any agreed language, within the constraints of the system's communication protocol, the approach may lead to a common improvement. Example languages are Knowledge Query Manipulation Language (KQML) or FIPA's Agent Communication Language (ACL).

Multi-agent system basics

Multiple agent systems paradigms

Many MAS systems are implemented in computer simulations, stepping the system through discrete "time steps". The MAS components communicate typically using a weighted request matrix, e.g.

```
 Speed-VERY_IMPORTANT: min=45 mph,
 Path length-MEDIUM_IMPORTANCE: max=60 expectedMax=40,
 Max-Weight-UNIMPORTANT
 Contract Priority-REGULAR
```

and a weighted response matrix, e.g.

```
 Speed-min:50 but only if weather sunny,
 Path length:25 for sunny / 46 for rainy
 Contract Priority-REGULAR
 note - ambulance will override this priority and you'll have to wait
```

A challenge-response-contract scheme is common in MAS systems, where

```
 First a "Who can?" question is distributed.
 Only the relevant components respond: "I can, at this price".
 Finally, a contract is set up, usually in several more short
communication steps between sides,
```

also considering other components, evolving "contracts", and the restriction sets of the component algorithms.

Another paradigm commonly used with MAS systems is the pheromone, where components "leave" information for other components "next in line" or "in the vicinity". These "pheromones" may "evaporate" with time, that is their values may decrease (or increase) with time.

Properties

MAS systems, also referred to as "self-organized systems", tend to find the best solution for their problems "without intervention". There is high similarity here to physical phenomena, such as energy minimizing, where physical objects tend to reach the lowest energy possible, within the physical constrained world. For example: many of the cars entering a metropolis in the morning, will be available for leaving that same metropolis in the evening.

The main feature which is achieved when developing multi-agent systems, if they work, is flexibility, since a multi-agent system can be added to, modified and reconstructed, without the need for detailed rewriting of the application. These systems also tend to be rapidly self-recovering and failure proof, usually due to the heavy redundancy of components and the self managed features, referred to, above.

The study of multi-agent systems

The study of multi-agent systems is "concerned with the development and analysis of sophisticated AI problem-solving and control architectures for both single-agent and multiple-agent systems." Topics of research in MAS include:

- agent-oriented software engineering
- beliefs, desires, and intentions (BDI)
- cooperation and coordination
- organization
- communication
- negotiation
- distributed problem solving
- multi-agent learning
- scientific communities
- dependability and fault-tolerance

Frameworks

While ad hoc multi-agent systems are often created from scratch by researchers and developers, some frameworks have arisen that implement common standards (such as the FIPA agent system platforms and communication languages). These frameworks save developers time and also aid in the standardization of MAS development.

Applications in the real world

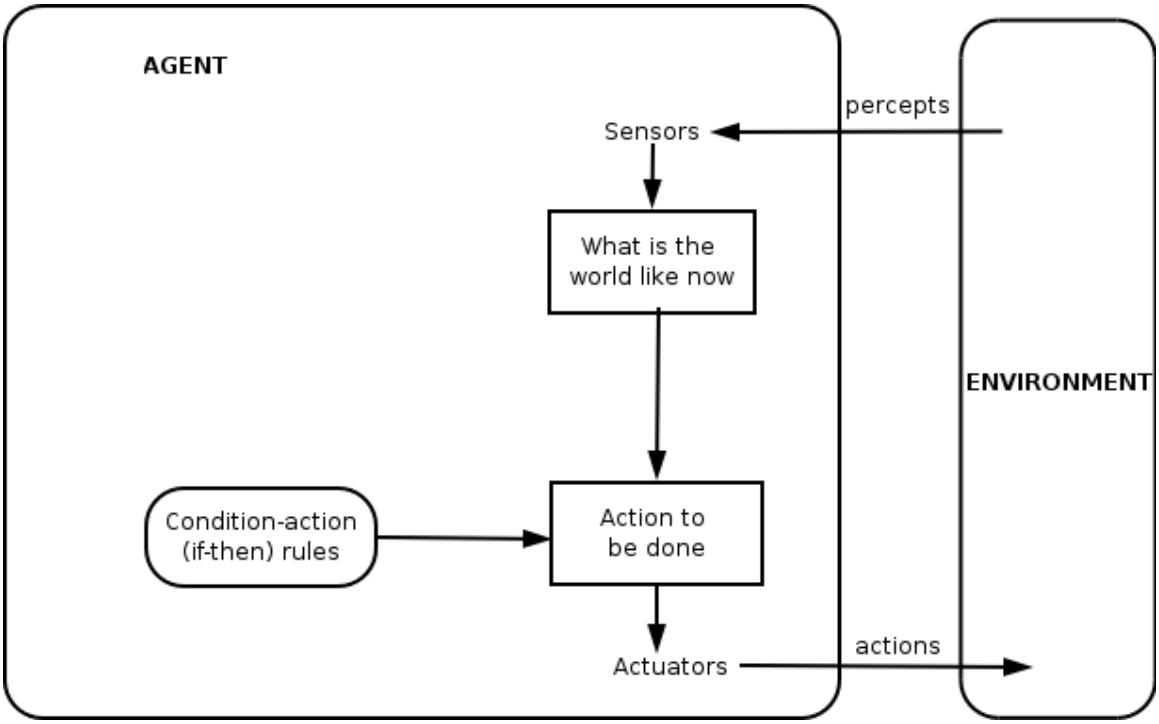
Multi-agent systems are applied in the real world to graphical applications such as computer games. Agent systems have been used in films. They are also used for coordinated defence systems. Other applications include transportation, logistics, graphics, GIS as well as in many other fields. It is widely being advocated for use in

networking and mobile technologies, to achieve automatic and dynamic load balancing, high scalability, and self-healing networks.

Chapter 2

Intelligent Agent

In artificial intelligence, an **intelligent agent** (**IA**) is an autonomous entity which observes and acts upon an environment (i.e. it is an agent) and directs its activity towards achieving goals (i.e. it is rational). Intelligent agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex: a reflex machine such as a thermostat is an intelligent agent, as is a human being, as is a community of human beings working together towards a goal.



Simple reflex agent

Intelligent agents are often described schematically as an abstract functional system similar to a computer program. For this reason, intelligent agents are sometimes called **abstract intelligent agent**s (AIA) to distinguish them from their real world implementations as computer systems, biological systems, or organizations. Some definitions of intelligent agents emphasize their autonomy, and so prefer the term **autonomous intelligent agent**s. Still others (notably Russell & Norvig (2003))

considered goal-directed behavior as the essence of intelligence and so prefer a term borrowed from economics, "rational agent".

Intelligent agents in artificial intelligence are closely related to agents in economics, and versions of the intelligent agent paradigm are studied in cognitive science, ethics, the philosophy of practical reason, as well as in many interdisciplinary socio-cognitive modeling and computer social simulations.

Intelligent agents are also closely related to software agents (an autonomous software program that carries out tasks on behalf of users). In computer science, the term *intelligent agent* may be used to refer to a software agent that has some intelligence, regardless if it is not a rational agent by Russell and Norvig's definition. For example, autonomous programs used for operator assistance or data mining (sometimes referred to as *bots*) are also called "intelligent agents".

A variety of definitions

Intelligent agents have been defined many different ways. According to Nikola Kasabov IA systems should exhibit the following characteristics:

- accommodate new problem solving rules incrementally
- adapt online and in real time
- be able to analyze itself in terms of behavior, error and success.
- learn and improve through interaction with the environment (embodiment)
- learn quickly from large amounts of data
- have memory-based exemplar storage and retrieval capacities
- have parameters to represent short and long term memory, age, forgetting, etc.

Structure of agents

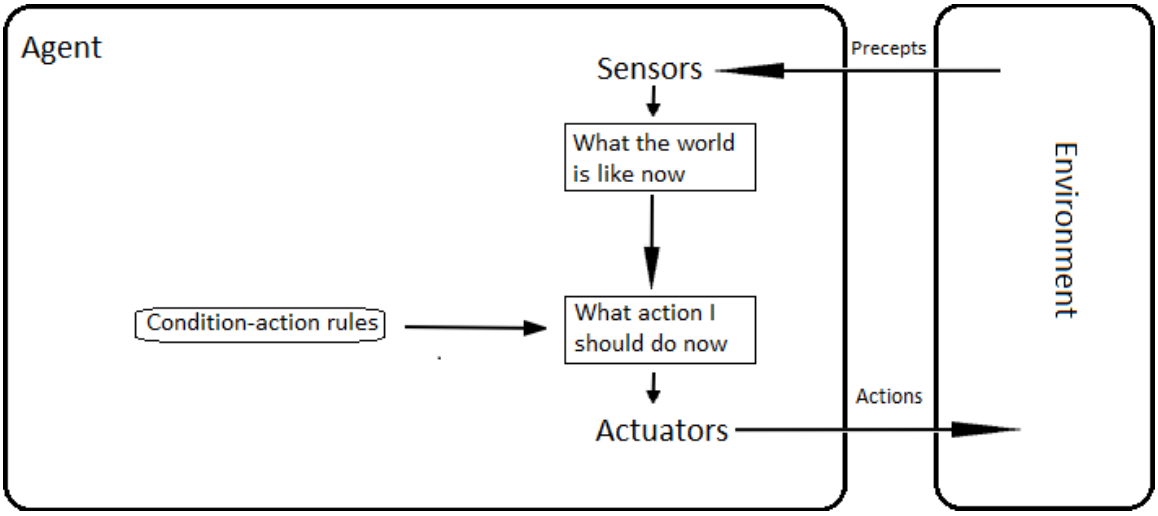
A simple agent program can be defined mathematically as an agent function which maps every possible percepts sequence to a possible action the agent can perform or to a coefficient, feedback element, function or constant that affects eventual actions:

$$f : P^* \rightarrow A$$

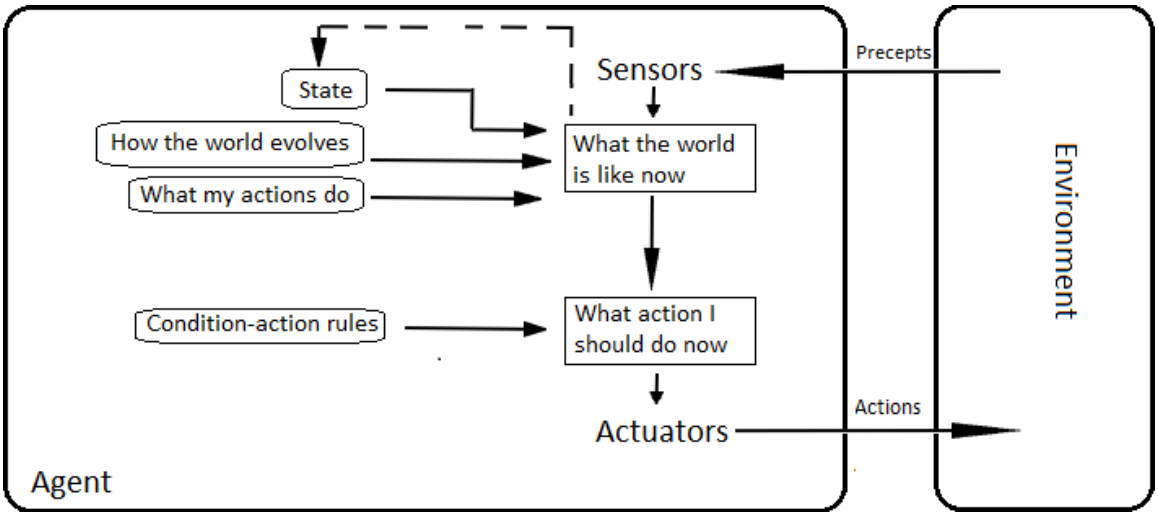
The program agent, instead, maps every possible percept to an action.

We use the term percept to refer to the agent's perceptional inputs at any given instant. In the following figures an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

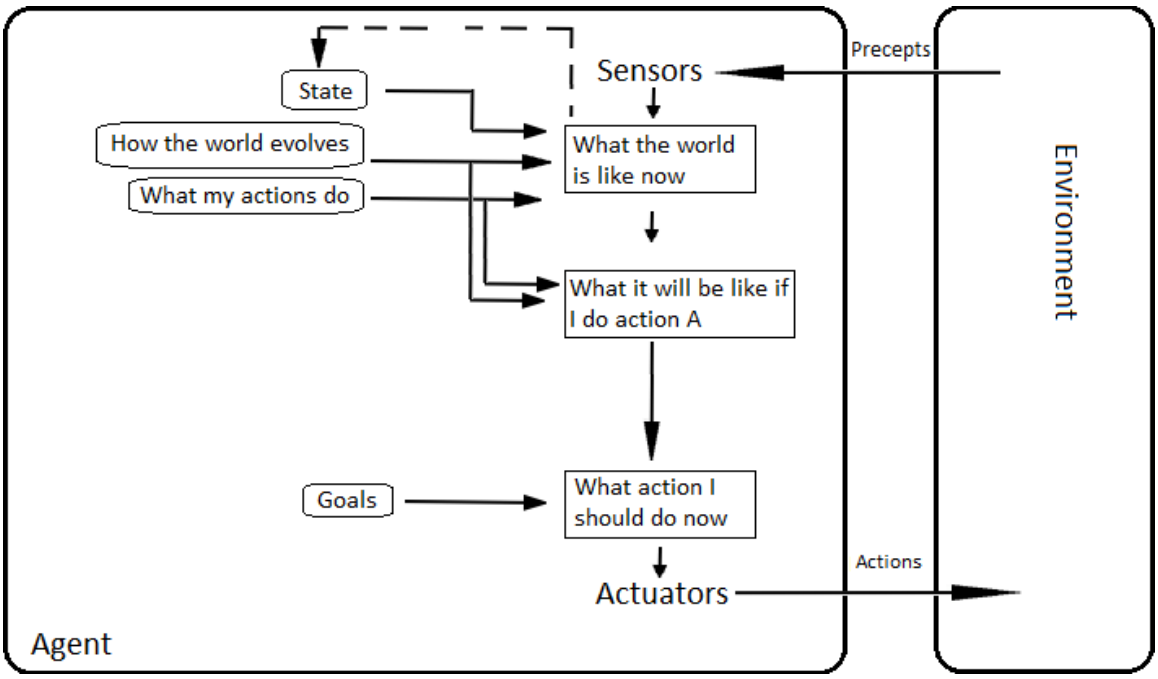
Classes of intelligent agents



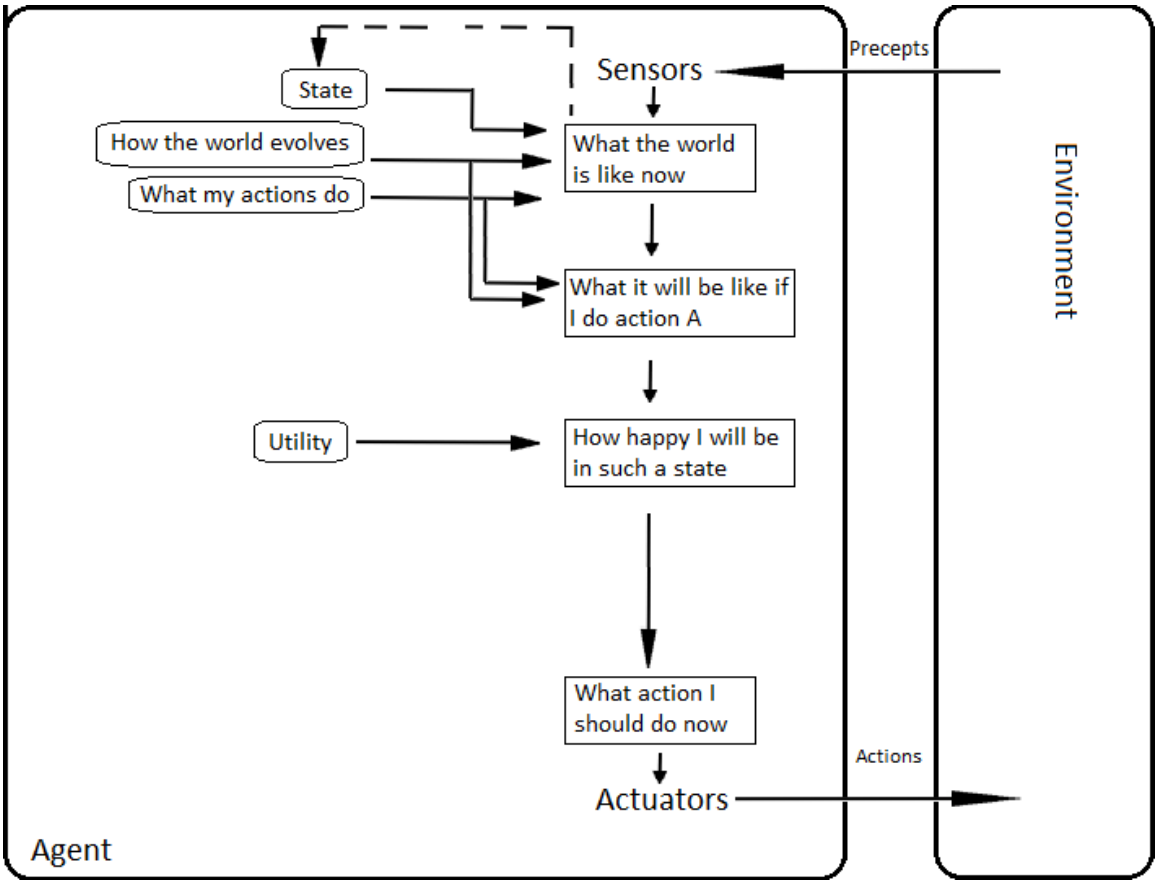
Simple reflex agent



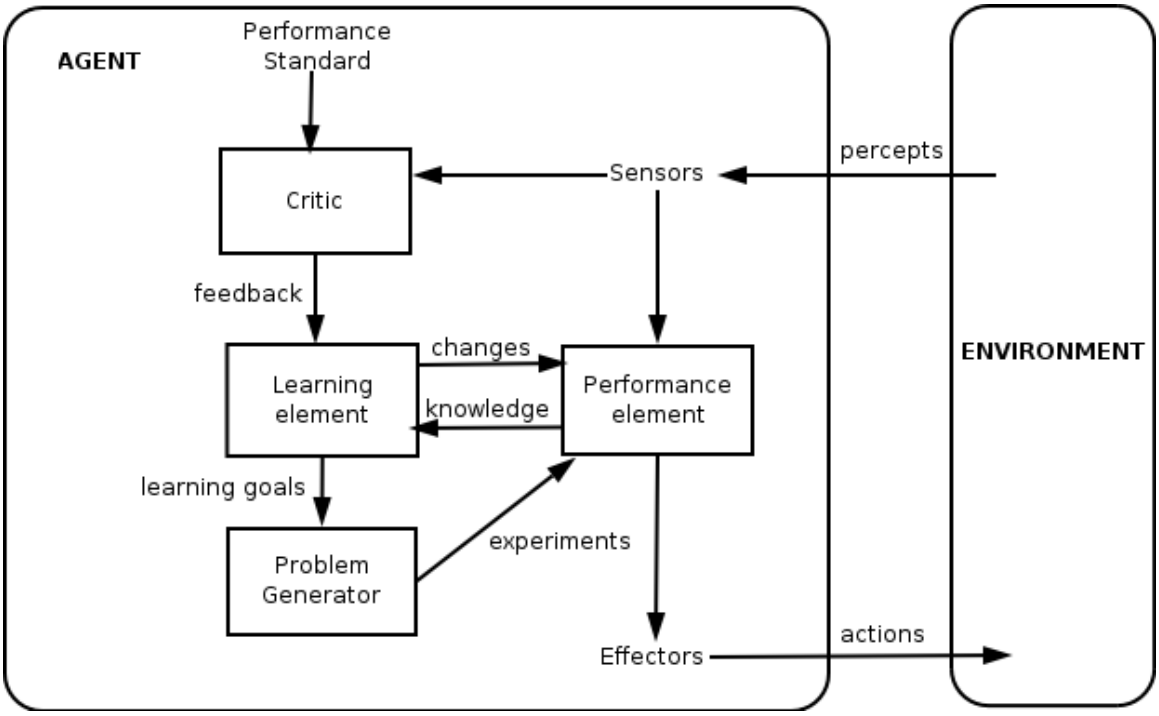
Model-based reflex agent



Model-based, goal-based agent



Model-based, utility-based agent



A general learning agent

Russell & Norvig (2003) group agents into five classes based on their degree of perceived intelligence and capability:

1. simple reflex agents
2. model-based reflex agents
3. goal-based agents
4. utility-based agents
5. learning agents

Simple reflex agents

Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the *condition-action rule*: if condition then action.

This agent function only succeeds when the environment is fully observable. Some reflex agents can also contain information on their current state which allows them to disregard conditions whose actuators are already triggered.

Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments. Note: If the agent can randomize its actions, it may be possible to escape from infinite loops.

Model-based reflex agents

A model-based agent can handle a partially observable environment. Its current state is stored inside the agent maintaining some kind of structure which describes the part of the world which cannot be seen. This knowledge about "how the world works" is called a model of the world, hence the name "model-based agent".

A model-based reflex agent should maintain some sort of internal model that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. It then chooses an action in the same way as the reflex agent.

Goal-based agents

Goal-based agents further expand on the capabilities of the model-based agents, by using "goal" information. Goal information describes situations that are desirable. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. Search and planning are the subfields of artificial intelligence devoted to finding action sequences that achieve the agent's goals.

In some instances the goal-based agent appears to be less efficient; it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

Utility-based agents

Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is. This measure can be obtained through the use of a *utility function* which maps a state to a measure of the utility of the state. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. The term utility, can be used to describe how "happy" the agent is.

A rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes- that is, the agent expects to derive, on average, given the probabilities and utilities of each outcome. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning.

Learning agents

Learning has an advantage that it allows the agents to initially operate in unknown environments and to become more competent than its initial knowledge alone might allow. The most important distinction is between the "learning element", which is responsible for making improvements, and the "performance element", which is responsible for selecting external actions.

The learning element uses feedback from the "critic" on how the agent is doing and determines how the performance element should be modified to do better in the future. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

The last component of the learning agent is the "problem generator". It is responsible for suggesting actions that will lead to new and informative experiences.

Other classes of intelligent agents

According to other sources, some of the sub-agents (not already mentioned in this treatment) that may be a part of an Intelligent Agent or a complete Intelligent Agent in themselves are:

- Decision Agents (that are geared to decision making);
- Input Agents (that process and make sense of sensor inputs - e.g. neural network based agents);
- Processing Agents (that solve a problem like speech recognition);
- Spatial Agents (that relate to the physical real-world);
- World Agents (that incorporate a combination of all the other classes of agents to allow autonomous behaviors).
- Believable agents - An agent exhibiting a personality via the use of an artificial character (the agent is embedded) for the interaction.
- Physical Agents - A physical agent is an entity which *percepts* through sensors and *acts* through actuators.
- Temporal Agents - A temporal agent may use time based stored information to offer instructions or data *acts* to a computer program or human being and takes program inputs *percepts* to adjust its next behaviors.

Hierarchies of agents

To actively perform their functions, Intelligent Agents today are normally gathered in a hierarchical structure containing many “sub-agents”. Intelligent sub-agents process and perform lower level functions. Taken together, the intelligent agent and sub-agents create a complete system that can accomplish difficult tasks or goals with behaviors and responses that display a form of intelligence.

Chapter 3

Agent-Based Model

An **agent-based model (ABM)** (also sometimes related to the term **multi-agent system** or **multi-agent simulation**) is a class of computational models for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) with a view to assessing their effects on the system as a whole. It combines elements of game theory, complex systems, emergence, computational sociology, multi-agent systems, and evolutionary programming. Monte Carlo Methods are used to introduce randomness. ABMs are also called individual-based models.

The models simulate the simultaneous operations and interactions of multiple agents, in an attempt to re-create and predict the appearance of complex phenomena. The process is one of emergence from the lower (micro) level of systems to a higher (macro) level. As such, a key notion is that simple behavioral rules generate complex behavior. This principle, known as K.I.S.S. ("Keep it simple stupid") is extensively adopted in the modeling community. Another central tenet is that the whole is greater than the sum of the parts. Individual agents are typically characterized as boundedly rational, presumed to be acting in what they perceive as their own interests, such as reproduction, economic benefit, or social status, using heuristics or simple decision-making rules. ABM agents may experience "learning", adaptation, and reproduction.

Most agent-based models are composed of: (1) numerous agents specified at various scales (typically referred to as agent-granularity); (2) decision-making heuristics; (3) learning rules or adaptive processes; (4) an interaction topology; and (5) a non-agent environment.

History

The idea of agent-based modeling was developed as a relatively simple concept in the late 1940s. Since it requires computation-intensive procedures, it did not become widespread until the 1990s.

The history of the agent-based model can be traced back to the Von Neumann machine, a theoretical machine capable of reproduction. The device von Neumann proposed would follow precisely detailed instructions to fashion a copy of itself. The concept was then improved by von Neumann's friend Stanisław Ulam, also a mathematician; Ulam suggested that the machine be built on paper, as a collection of cells on a grid. The idea intrigued von Neumann, who drew it up—creating the first of the devices later termed cellular automata.

Another improvement was introduced by the mathematician John Conway. He constructed the well-known Game of Life. Unlike von Neumann's machine, Conway's Game of Life operated by tremendously simple rules in a virtual world in the form of a 2-dimensional checkerboard.

One of the earliest agent-based models in concept was Thomas Schelling's segregation model, which was discussed in his paper *Dynamic Models of Segregation* in 1971. Though Schelling originally used coins and graph paper rather than computers, his models embodied the basic concept of agent-based models as autonomous agents interacting in a shared environment with an observed aggregate, emergent outcome.

In the early 1980s, Robert Axelrod hosted a tournament of Prisoner's Dilemma strategies and had them interact in an agent-based manner to determine a winner. Axelrod would go on to develop many other agent-based models in the field of political science that examine phenomena from ethnocentrism to the dissemination of culture (Axelrod 1997).

In the late 1980s, Craig Reynolds' work on flocking models contributed to the development of some of the first biological agent-based models that contained social characteristics. He tried to model the reality of lively biological agents, known as artificial life, a term coined by Christopher Langton.

The first use of the word "agent" and a definition as it is currently used today is hard to track down. One candidate appears to be John Holland and John H. Miller's 1991 paper "Artificial Adaptive Agents in Economic Theory" which is based on an earlier conference presentation of theirs.

At the same time, during the 1980s, social scientists, mathematicians, operations researchers, and a scattering of people from other disciplines developed Computational and Mathematical Organization Theory (CMOT). This field grew as a special interest group of The Institute of Management Sciences (TIMS) and its sister society, the Operations Research Society of America (ORSA). Through the mid-1990s, the field focused on such issues as designing effective teams, understanding the communication required for organizational effectiveness, and the behavior of social networks. With the appearance of StarLogo in 1990, SWARM and NetLogo in the mid-1990s and RePast in 2000, as well as some custom-designed code, CMOT—later renamed Computational Analysis of Social and Organizational Systems (CASOS) -- incorporated more and more agent-based modeling. Samuelson (2000) is a good brief overview of the early history, and Samuelson (2005) and Samuelson and Macal (2006) trace the more recent

developments. Bonabeau (2002) is a good survey of the potential of agent-based modeling as of the time that its modelling software became widely available.

Kathleen M. Carley developed an early ABM, Construct , to explore the co-evolution of social networks and culture.

Joshua M. Epstein and Robert Axtell developed a large-scale ABM, the Sugarscape, to simulate and explore the role of social phenomenon such as seasonal migrations, pollution, sexual reproduction, combat, and transmission of disease and even culture.

Nigel Gilbert published the first textbook on Social Simulation: Simulation for the social scientist (1999) and established its most relevant journal: the Journal of Artificial Societies and Social Simulation.

In the late 1990s, the merger of TIMS and ORSA to form INFORMS, and the move by INFORMS from two meetings each year to one, helped to spur the CMOT group to form a separate society, the North American Association for Computational Social and Organizational Sciences (NAACSOS). Kathleen Carley, of Carnegie Mellon University, was a major contributor, especially to models of social networks, obtaining National Science Foundation funding for the annual conference and serving as the first President of NAACSOS. She was succeeded by David Sallach of the University of Chicago and Argonne National Laboratory, and then by Michael Prietula of Emory University. At about the same time NAACSOS began, the European Social Simulation Association (ESSA) and the Pacific Asian Association for Agent-Based Approach in Social Systems Science (PAAA), counterparts of NAACSOS, were organized. Nowadays, these three organizations collaborate internationally. The First World Congress on Social Simulation was held under their joint sponsorship in Kyoto, Japan, in August 2006. The Second World Congress was held in the northern Virginia suburbs of Washington, D.C., in July 2008, with George Mason University taking the lead role in local arrangements.

More recently, Ron Sun developed methods for basing agent-based simulation on models of human cognition, known as cognitive social simulation. Bill McKelvey, Suzanne Lohmann, Dario Nardi, Dwight Read and others at UCLA have also made significant contributions in organizational behavior and decision-making. Since 2001, UCLA has arranged a conference at Lake Arrowhead, California, that has become another major gathering point for practitioners in this field.

Theory

Most computational modeling research describes systems in equilibrium or as moving between equilibria. Agent-based modeling, however, using simple rules, can result in far more complex and interesting behavior.

The three ideas central to agent-based models are agents as objects, emergence, and complexity.

Agent-based models consist of dynamically interacting rule-based agents. The systems within which they interact can create real world-like complexity. These agents are:

- Intelligent and purposeful.
- Situated in space and time. They reside in networks and in lattice-like neighborhoods. The location of the agents and their responsive and purposeful behavior are encoded in algorithmic form in computer programs. The modeling process is best described as inductive. The modeler makes those assumptions thought most relevant to the situation at hand and then watches phenomena emerge from the agents' interactions. Sometimes that result is an equilibrium. Sometimes it is an emergent pattern. Sometimes, however, it is an unintelligible mangle.

In some ways, agent-based models complement traditional analytic methods. Where analytic methods enable humans to characterize the equilibria of a system, agent-based models allow the possibility of generating those equilibria. This generative contribution may be the most mainstream of the potential benefits of agent-based modeling. Agent-based models can explain the emergence of higher order patterns—network structures of terrorist organizations and the Internet, power law distributions in the sizes of traffic jams, wars, and stock market crashes, and social segregation that persists despite populations of tolerant people. Agent-based models also can be used to identify lever points, defined as moments in time in which interventions have extreme consequences, and to distinguish among types of path dependency.

Rather than focusing on stable states, the models consider a system's robustness—the ways that complex systems adapt to internal and external pressures so as to maintain their functionalities. The task of harnessing that complexity requires consideration of the agents themselves—their diversity, connectedness, and level of interactions.

Applications

Agent-based models have been used since the mid-1990s to solve a variety of business and technology problems. Examples of applications include supply chain optimization and logistics, modeling of consumer behavior, including word of mouth, social network effects, distributed computing, workforce management, and portfolio management. They have also been used to analyze traffic congestion. In these and other applications, the system of interest is simulated by capturing the behavior of individual agents and their interconnections. Agent-based modeling tools can be used to test how changes in individual behaviors will affect the system's emerging overall behavior.

Other models have analyzed the spread of epidemics, the threat of biowarfare, biological applications including population dynamics, the growth and decline of ancient civilizations, evolution of ethnocentric behavior, forced displacement/migration, language choice dynamics, and biomedical applications including inflammation and the human immune system. Agent-based models have also been used for developing decision support systems such as for breast cancer.

Recently, agent based modelling and simulation has been applied to various domains such as studying the impact of publication venues by researchers in the computer science domain (journals versus conferences). In addition, ABMS has been used to simulate information delivery in ambient assisted environments. In the domain of peer-to-Peer, ad-hoc and other self-organizing and complex networks, the usefulness of agent based modeling and simulation has been shown. The use of Computer Science based Formal Specification framework coupled with Wireless sensor networks and an Agent-based simulation has recently been demonstrated in.

Agent based evolutionary search or algorithm is a new research topic for solving complex optimization problems. Further details on the topic can be found in R. Sarker and T. Ray (2010) Agent based Evolutionary Approach: An Introduction, Agent Based Evolutionary Search, Springer series in Evolutionary Learning and Optimization, Springer, pp. 1–12.

Hardware

The Software described above is designed for serial von-Neumann computer architectures. This limits the speed and scalability of these systems. A recent development is the use of data-parallel algorithms on Graphics Processing Units GPUs for ABM simulation , and . The extreme memory bandwidth combined with the sheer number crunching power of multi-processor GPUs has enabled simulation of millions of agents at tens of frames per second.

Verification & Validation of Agent-Based Models

Verification and validation of simulation models is extremely important. Verification involves debugging the model to ensure it works correctly; whereas Validation ensures that you have built the right model. Verification and validation in the social sciences domain can be seen in. In Computational Economics, validation can be examined in. In, the author proposes face validation, sensitivity analysis, calibration and statistical validation. Discrete-Event Simulation Framework approach for the validation of Agent-Based systems has been proposed in. A comprehensive resource on empirical validation of agent-based models is

A formal approach for V&V of all agent-based models is based on building a VOMAS (Virtual Overlay Multi-Agent System), a software engineering based approach, where a virtual overlay Multi-agent system is developed alongside the agent-based model. The agents in the Multi-Agent System are able to gather data by generation of logs as well as provide run-time validation and verification support by watch agents and also agents to check any violation of invariants at run-time. These are set by the Simulation Specialist with help from the SME (Subject Matter Expert). An example of using VOMAS for Verification and Validation of a Forest Fire simulation model is given in

VOMAS provides a formal way of Validation and Verification. If you want to build a VOMAS, you need to start designing VOMAS agents along with the agents in the actual

simulation preferably from the start. So, in essence, by the time your simulation model is complete, you essentially have one model which contains two models:

1. An Agent Based Model of the intended system
2. An Agent Based Model of the VOMAS

Unlike all previous work on Verification and Validation, VOMAS agents ensure that the simulations are validated in-simulation i.e. even during execution. In case of any exceptional situations, which are programmed on the directive of the Simulation Specialist (SS), the VOMAS agents can report them. In addition, the VOMAS agents can be used to log key events for the sake of debugging and subsequent analysis of simulations. In other words, VOMAS allows for a flexible use of any given technique for the sake of Verification and Validation of an Agent-based Model in any domain.

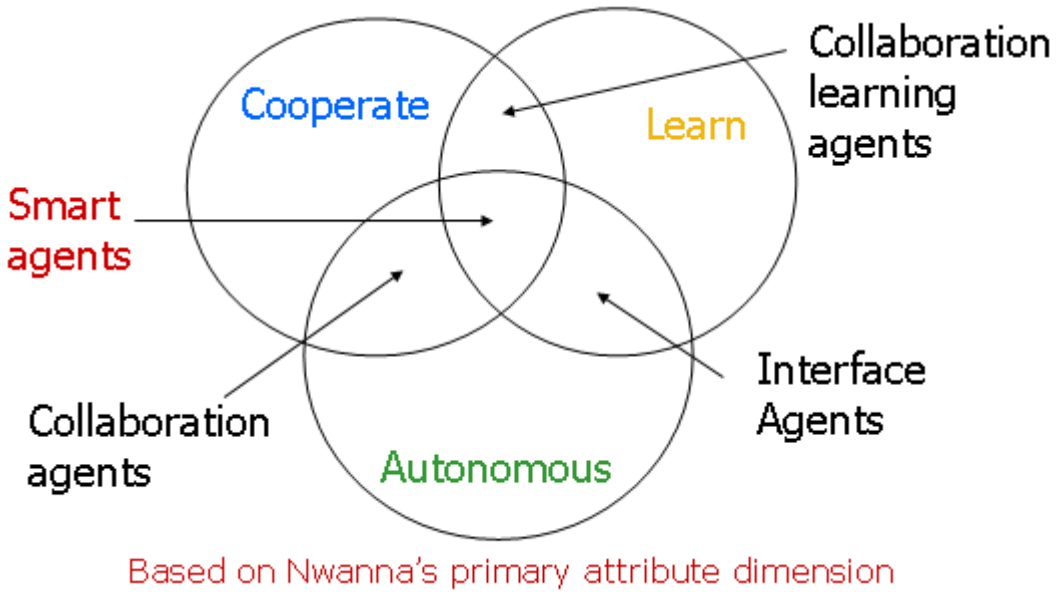
Chapter 4

Software Agent

In computer science, a **software agent** is a piece of software that acts for a user or other program in a relationship of agency, which derives from the Latin *agere* (to do): an agreement to act on one's behalf. Such "action on behalf of" implies the authority to decide which (and if) action is appropriate. The idea is that agents are not strictly invoked for a task, but activate themselves.

Related and derived concepts include *Intelligent agents* (in particular exhibiting some aspect of Artificial Intelligence, such as learning and reasoning), *autonomous* agents (capable of modifying the way in which they achieve their objectives), *distributed* agents (being executed on physically distinct computers), *multi-agent systems* (distributed agents that do not have the capabilities to achieve an objective alone and thus must communicate), and *mobile* agents (agents that can relocate their execution onto different processors).

Definition



Nwana's Category of Software Agent

The term "agent" describes a software abstraction, an idea, or a concept, similar to OOP terms such as methods, functions, and objects. The concept of an agent provides a convenient and powerful way to describe a complex software entity that is capable of acting with a certain degree of autonomy in order to accomplish tasks on behalf of its user. But unlike objects, which are defined in terms of *methods* and *attributes*, an agent is defined in terms of its behavior.

Various authors have proposed different definitions of agents, these commonly include concepts such as

- *persistence* (code is not executed on demand but runs continuously and decides for itself when it should perform some activity)
- *autonomy* (agents have capabilities of task selection, prioritization, goal-directed behaviour, decision-making without human intervention)
- *social ability* (agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task)
- *reactivity* (agents perceive the context in which they operate and react to it appropriately).

What an agent is not

It is not useful to prescribe what is, and what is not an agent. However contrasting the term with related concepts may help clarify its meaning:

Distinguishing agents from programs

Franklin & Graesser (1997) discuss four key notions that distinguish agents from arbitrary programs: reaction to the environment, autonomy, goal-orientation and persistence. Related and derived concepts include Intelligent agents (in particular exhibiting some aspect of Artificial Intelligence, such as learning and reasoning), autonomous agents (capable of modifying the way in which they achieve their objectives), distributed agents (being executed on physically distinct computers), multi-agent systems (distributed agents that do not have the capabilities to achieve an objective alone and thus must communicate), and mobile agents (agents that can relocate their execution onto different processors).

Intuitive distinguishing agents from objects

- Agents are more autonomous than objects.
- Agents have flexible behaviour, reactive, proactive, social.
- Agents have at least one thread of control but may have more.

(Wooldridge, 2002)

Distinguishing agents from expert systems

- Expert systems are not coupled to their environment;
- Expert systems are not designed for reactive, proactive behavior.
- Expert systems do not consider social ability

  (Wooldridge, 2003)

Distinguishing intelligent software agents from intelligent agents in artificial intelligence

- Intelligent agents (also known as rational agents) are not just software programs, they may also be machines, human beings, communities of human beings (such as firms) or anything that is capable of goal directed behavior.

  (Russell & Norvig 2003)

History

The concept of an agent can be traced back to Hewitt's Actor Model (Hewitt, 1977) - "A self-contained, interactive and concurrently-executing object, possessing internal state and communication capability."

To be more academic, software agent systems are a direct evolution from Multi-Agent Systems (MAS). MAS evolved from Distributed Artificial Intelligence (DAI), Distributed Problem Solving (DPS) and Parallel AI (PAI), thus inheriting all characteristics (good and bad) from DAI and AI.

John Sculley's 1987 "Knowledge Navigator" video portrayed an image of a relationship between end-users and agents. Being an ideal first, this field experienced a series of unsuccessful top-down implementations, instead of a piece-by-piece, bottom-up approach. The range of agent types is now (from 1990) broad: WWW, search engines, etc.

Examples

Intelligent software agents

Haag (2006) suggests that there are only four essential types of intelligent software agents:

1. Buyer agents or shopping bots
2. User or personal agents
3. Monitoring-and-surveillance agents
4. Data Mining agents

Buyer agents (shopping bots)

Buyer agents travel around a network (i.e. the internet) retrieving information about goods and services. These agents, also known as 'shopping bots', work very efficiently for commodity products such as CDs, books, electronic components, and other one-size-fits-all products.

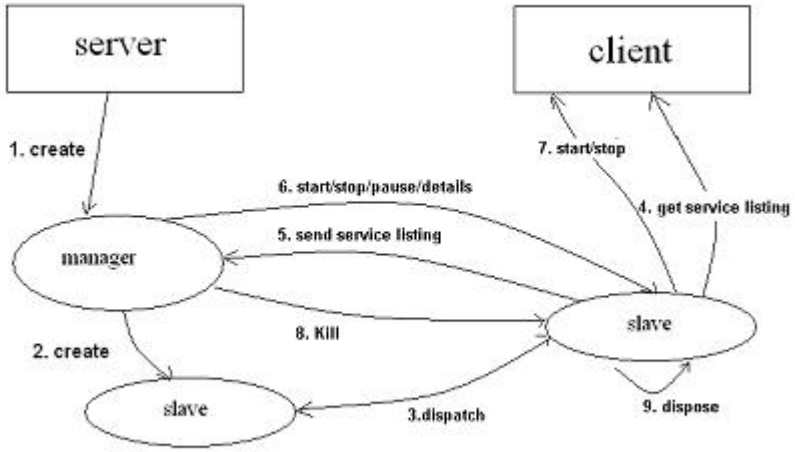
User agents (personal agents)

User agents, or personal agents, are intelligent agents that take action on your behalf. In this category belong those intelligent agents that already perform, or will shortly perform, the following tasks:

- Check your e-mail, sort it according to the user's order of preference, and alert you when important emails arrive.
- Play computer games as your opponent or patrol game areas for you.
- Assemble customized news reports for you. There are several versions of these, including CNN.
- Find information for you on the subject of your choice.
- Fill out forms on the Web automatically for you, storing your information for future reference
- Scan Web pages looking for and highlighting text that constitutes the "important" part of the information there
- "Discuss" topics with you ranging from your deepest fears to sports
- Facilitate with online job search duties by scanning known job boards and sending the resume to opportunities who meet the desired criteria
- Profile synchronization across heterogeneous social networks

Monitoring-and-surveillance (predictive) agents

Monitoring and Surveillance Agents are used to observe and report on equipment, usually computer systems. The agents may keep track of company inventory levels, observe competitors' prices and relay them back to the company, watch stock manipulation by insider trading and rumors, etc.



service monitoring

For example, NASA's Jet Propulsion Laboratory has an agent that monitors inventory, planning, and scheduling equipment ordering to keep costs down, as well as food storage facilities. These agents usually monitor complex computer networks that can keep track of the configuration of each computer connected to the network.

A special case of Monitoring-and-Surveillance agents are organizations of agents used to emulate the Human Decision Making process during tactical operations. The agents monitor the status of assets (ammunition, weapons available, platforms for transport, etc.) and receive Goals (Missions) from higher level agents. The Agents then pursue the Goals with the Assets at hand, minimizing expenditure of the Assets while maximizing Goal Attainment.

Data mining agents

This agent uses information technology to find trends and patterns in an abundance of information from many different sources. The user can sort through this information in order to find whatever information they are seeking.

A data mining agent operates in a data warehouse discovering information. A 'data warehouse' brings together information from lots of different sources. "Data mining" is the process of looking through the data warehouse to find information that you can use to take action, such as ways to increase sales or keep customers who are considering defecting.

'Classification' is one of the most common types of data mining, which finds patterns in information and categorizes them into different classes. Data mining agents can also detect major shifts in trends or a key indicator and can detect the presence of new

information and alert you to it. For example, the agent may detect a decline in the construction industry for an economy; based on this relayed information construction companies will be able to make intelligent decisions regarding the hiring/firing of employees or the purchase/lease of equipment in order to best suit their firm.

Other examples

Some other examples of current Intelligent agents include some spam filters, game bots, and server monitoring tools. Search engine indexing bots also qualify as intelligent agents.

- User agent - for browsing the World Wide Web
- Mail transfer agent - For serving E-mail, such as *Microsoft Outlook*. Why? It communicates with the POP3 mail server, without users having to understand POP3 command protocols. It even has rule sets that filter mail for the user, thus sparing them the trouble of having to do it themselves.
- SNMP agent
- DAML (DARPA Agent Markup Language)
- Jason (multi-agent systems development platform)
- 3APL (Artificial Autonomous Agents Programming Language)
- GOAL Agent Programming Language
- Web Ontology Language (OWL)
- *daemons* in Unix-like systems.
- In Unix-style networking servers, *httpd* is an HTTP daemon which implements the HyperText Transfer Protocol at the root of the World Wide Web
- Management agents used to manage telecom devices
- Crowd simulation for safety planning or 3D computer graphics,
- Java Agent Template (JAT)

Design issues

Interesting issues to consider in the development of agent-based systems include

- how tasks are scheduled and how synchronization of tasks is achieved
- how tasks are prioritized by agents
- how agents can collaborate, or recruit resources,
- how agents can be re-instantiated in different environments, and how their internal state can be stored,
- how the environment will be probed and how a change of environment leads to behavioral changes of the agents
- how messaging and communication can be achieved,
- what hierarchies of agents are useful (e.g. task execution agents, scheduling agents, resource providers ...).

For software agents to work together efficiently they must share semantics of their data elements. This can be done by having computer systems publish their metadata.

The definition of *agent processing* can be approached from two interrelated directions:

- internal state processing and ontologies for representing knowledge
- interaction protocols - standards for specifying communication of tasks

Agent systems are used to model real world systems with concurrency or parallel processing.

- Agent Machinery - Engines of various kinds, which support the varying degrees of intelligence
- Agent Content - Data employed by the machinery in Reasoning and Learning
- Agent Access - Methods to enable the machinery to perceive content and perform actions as outcomes of Reasoning
- Agent Security - Concerns related to distributed computing, augmented by a few special concerns related to agents

The agent uses its access methods to go out into local and remote databases to forage for content. These access methods may include setting up news stream delivery to the agent, or retrieval from bulletin boards, or using a spider to walk the Web. The content that is retrieved in this way is probably already partially filtered – by the selection of the newsfeed or the databases that are searched. The agent next may use its detailed searching or language-processing machinery to extract keywords or signatures from the body of the content that has been received or retrieved. This abstracted content (or event) is then passed to the agent's Reasoning or inferencing machinery in order to decide what to do with the new content. This process combines the event content with the rule-based or knowledge content provided by the user. If this process finds a good hit or match in the new content, the agent may use another piece of its machinery to do a more detailed search on the content. Finally, the agent may decide to take an action based on the new content; for example, to notify the user that an important event has occurred. This action is verified by a security function and then given the authority of the user. The agent makes use of a user-access method to deliver that message to the user. If the user confirms that the event is important by acting quickly on the notification, the agent may also employ its learning machinery to increase its weighting for this kind of event.

Impacts of software agents

It is unarguable that software agents are innovative technologies that may offer various benefits to their end users by automating complex or repetitive tasks. However, there are several potential organizational and cultural impacts of this technology that need to be considered. Organizational impacts include the transformation of the entire electronic commerce sector, operational encumbrance, and security overload. Software agents are able to quickly search the Internet, identify the best offers available online, and present this information to the end users in aggregate form. Therefore, users may not need to manually browse various websites of individual merchants; they are able to locate the best deal in a matter of seconds. At the same time, this increases price-based competition and transforms the entire electronic commerce sector into a uniform perfect competition

market. The implementation of agents also requires additional resources from the companies, places an extra burden on their networks, and requires new security procedures. The cultural effects of the implementation of software agents include trust affliction, skills erosion, privacy attrition and social detachment. Some users may not feel entirely comfortable fully delegating important tasks to software applications. Those who start relying solely on intelligent agents may lose important skills, for example, relating to information literacy. In order to act on a user's behalf, a software agent needs to have a complete understanding of a user's profile, including his/her personal preferences. This, in turn, may lead to unpredictable privacy issues. When users start relying on their software agents more, especially, for communication activities, they may lose contact with other human users and look at the word with the eyes of their agents. It is these consequences that agent researchers and users need to consider dealing with intelligent agent technologies.

Chapter 5

GOAL Agent Programming Language

GOAL is an agent programming language for programming rational agents. GOAL agents derive their choice of action from their beliefs and goals. The language provides the basic building blocks to design and implement rational agents by means of a set of programming constructs. These programming constructs allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. The language provides an intuitive programming framework based on common sense or practical reasoning.

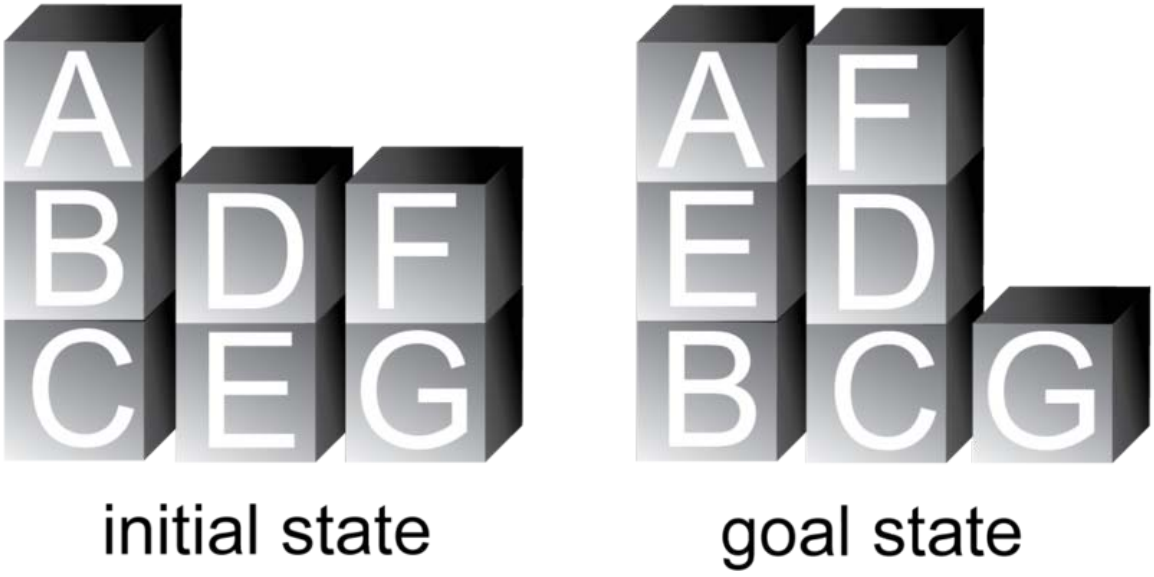
Overview

The main features of GOAL include:

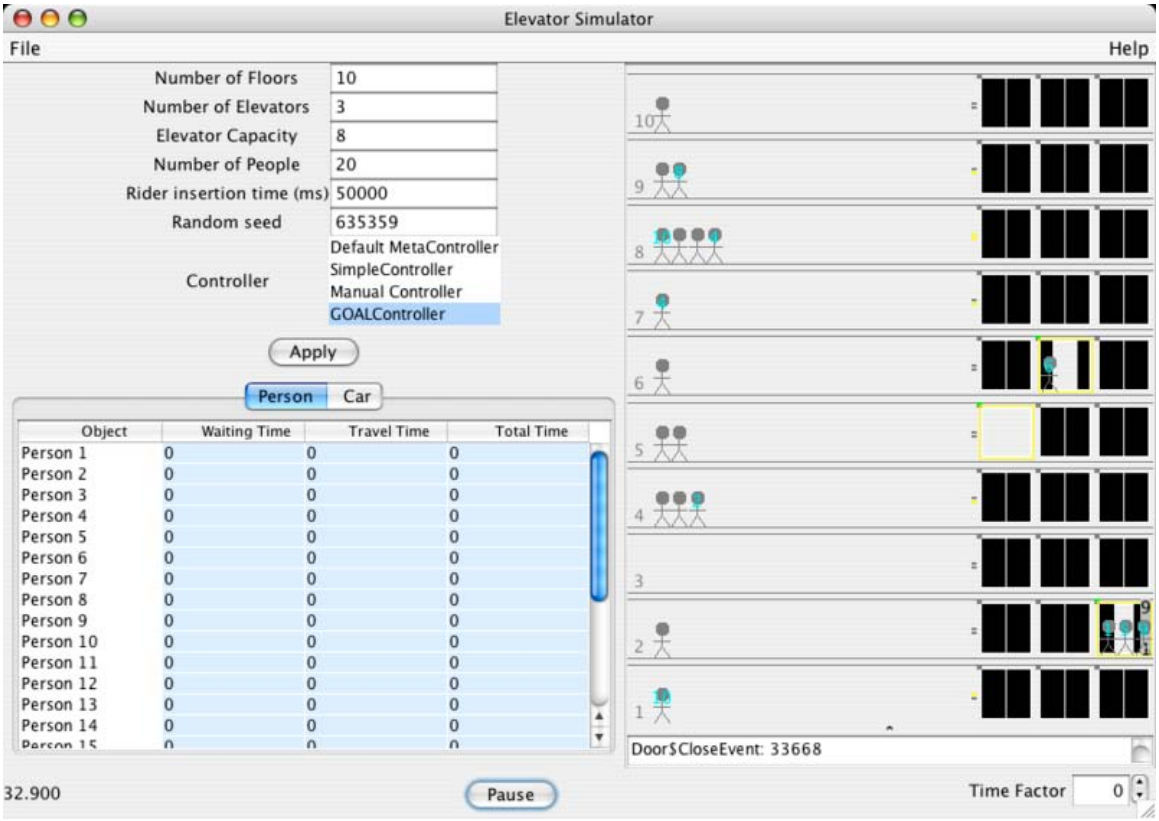
- **Declarative beliefs**: Agents use a symbolic, logical language to represent the information they have, and their beliefs or knowledge about the environment they act upon in order to achieve their goals. This *knowledge representation language* is not fixed by GOAL but, in principle, may be varied according to the needs of the programmer.
- **Declarative goals**: Agents may have multiple goals that specify *what* the agent wants to achieve at some moment in the near or distant future. Declarative goals specify a state of the environment that the agent wants to establish, they do not specify actions or procedures how to achieve such states.
- **Blind commitment strategy**: Agents commit to their goals and drop goals only when they have been achieved. This commitment strategy, called a *blind* commitment strategy in the literature, is the *default* strategy used by GOAL agents. Rational agents are assumed to not have goals that they believe are already achieved, a constraint which has been built into GOAL agents by dropping a goal when it has been *completely* achieved.
- **Rule-based action selection**: Agents use so-called *action rules* to select actions, given their beliefs and goals. Such rules may *underspecify* the choice of action in the sense that multiple actions may be performed at any time given the action rules of the agent. In that case, a GOAL agent will select an arbitrary enabled action for execution.

- **Policy-based intention modules**: Agents may focus their attention and put all their efforts on achieving a subset of their goals, using a subset of their actions, using only knowledge relevant to achieving those goals. GOAL provides modules to structure action rules and knowledge dedicated to achieving specific goals. Informally, modules can be viewed as policy-based intentions in the sense of Michael Bratman.
- **Communication at the knowledge level**: Agents may communicate with each other to exchange information, and to coordinate their actions. GOAL agents communicate using the knowledge representation language that is also used to represent their beliefs and goals.

GOAL Agent Program



An Example Blocks World Problem



Another Example: A GOAL Multi-Agent Elevator Controller

A GOAL agent program consists of six different sections, including the *knowledge*, *beliefs*, *goals*, *action rules*, *action specifications*, and *percept rules*, respectively. The knowledge, beliefs and goals are represented in a knowledge representation language such as Prolog, Answer set programming, SQL (or Datalog), or the Planning Domain Definition Language, for example. Below, we illustrate the components of a GOAL agent program using Prolog.

The overall structure of a GOAL agent program looks like:

```
main: <agentname> {
  <sections>
}
```

The GOAL agent code used to illustrate the structure of a GOAL agent is an agent that is able to solve Blocks world problems. The beliefs of the agent represent the current state of the Blocks world whereas the goals of the agent represent the goal state. The *knowledge* section listed next contains additional conceptual or domain knowledge related to the Blocks world domain.

```
knowledge{
  block(a), block(b), block(c), block(d), block(e), block(f), block(g).
  clear(table).
  clear(X) :- block(X), not(on(Y,X)).
```

```
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
```

Note that all the blocks listed in the knowledge section reappear in the *beliefs* section again as the position of each block needs to be specified to characterize the complete configuration of blocks.

```
beliefs{
  on(a,b), on(b,c), on(c,table), on(d,e), on(e,table), on(f,g),
on(g,table).
}
```

All known blocks also are present in the *goals* section which specifies a goal configuration which reuses all blocks.

```
goals{
  on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d),
on(g,table).
}
```

A GOAL agent may have multiple goals at the same time. These goals may even be conflicting as each of the goals may be realized at different times. For example, an agent might have a goal to watch a movie in the movie theater and to be at home (afterwards).

In GOAL, different notions of goal are distinguished. A **primitive goal** is a statement that follows from the goal base in conjunction with the concepts defined in the knowledge base. For example, `tower([a,e,b])` is a primitive goal and we write `goal(tower([a,e,b])` to denote this. Initially, `tower([a,e,b])` is also an **achievement goal** since the agent does not believe that a is on top of e, e is on top of b, and b is on the table. Achievement goals are primitive goals that the agent does not believe to be the case and are denoted by `a-goal(tower([a,e,b])`. It is also useful to be able to express that a **goal has been achieved**. `goal-a(tower([e,b])` is used to express, for example, that the tower `[e,b]` has been achieved with block e on top of block b. Both achievement goals as well as the notion of a goal achieved can be defined:

```
a-goal(formula) ::= goal(formula), not(bel(formula))
goal-a(formula) ::= goal(formula), bel(formula)
```

There is a significant literature on defining the concept of an achievement goal in the agent literature.

GOAL is a rule-based programming language. Rules are structured into modules. The *main module* of a GOAL agent specifies a strategy for selecting actions by means of action rules. The first rule below states that moving block X on top of block Y (or, possibly, the table) is an option if such a move is constructive, i.e. moves the block in position. The second rule states that moving a block X to the table is an option if block X is misplaced.

```
main module{
  program{
    if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then move(X,Y).
    if a-goal(tower([X|T])) then move(X,table).
  }
}
```

Actions, such as the move action used above, are specified using a STRIPS-style specification of preconditions and postconditions. A precondition specifies when the action can be performed (is enabled). A postcondition specifies what the effects of performing the action are.

```
actionspec{
  move(X,Y) {
    pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
    post{ not(on(X,Z)), on(X,Y) }
}
```

Finally, the *event module* consists of rules for processing events such as percepts received from the environment. The rule below specifies that for all percepts received that indicate that block X is on block Y, and X is believed to be on top of Z unequal to Y, the new fact on(X,Y) is to be added to the belief base and the atom on(X,Z) is to be removed.

```
event module{
  program{
    forall bel( percept(on(X,Y)), on(X,Z), not(Y=Z) ) do
insert(on(X,Y), not(on(X,Z))).
  }
}
```

Download

GOAL is available for download from the GOAL webpage hosted at the Delft University of Technology. Besides the GOAL installer the GOAL webpage provides the GOAL Programming Guide, GOAL IDE User Manual, and a Video Tutorial.

Related Agent Programming Languages

The GOAL agent programming language is related to but different from other agent programming languages such as AGENT0, AgentSpeak, 2APL, Golog, JACK Intelligent Agents, Jadex, and, for example, Jason. The distinguishing feature of GOAL are the concept of a declarative goal. Goals of a GOAL agent describe *what* an agent wants to achieve, not how to achieve it. Different from other languages, GOAL agents are committed to their goals and only remove a goal when it has been *completely* achieved. GOAL provides a programming framework with a strong focus on declarative programming and the reasoning capabilities required by rational agents.

Chapter 6

Deliberative Agent & Belief-Desire-Intention Software Model

Deliberative Agent

Deliberative agent (also known as intentional agent) is a sort of software agent used mainly in multi-agent system simulations. According to Wooldridge's definition, a deliberative agent is "one that possesses an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via symbolic reasoning".

Compared to reactive agents, which are able to reach their goal only by reacting reflexively on external stimuli, a deliberative agent's internal processes are more complex. The difference lies in fact, that deliberative agent maintains a symbolic representation of the world it inhabits. In other words, it possesses internal image of the external environment and is thus capable to plan its actions. Most commonly used architecture for implementing such behavior is Belief-Desire-Intention software model (BDI), where an agent's beliefs about the world (its image of a world), desires (goal) and intentions are internally represented and practical reasoning is applied to decide, which action to select.

There has been considerable research focused on integrating both reactive and deliberative agent strategies resulting in developing a compound called hybrid agent, which combines extensive manipulation with nontrivial symbolic structures and reflexive reactive responses to the external events.

How does deliberative agent work?

It has already been mentioned, that deliberative agents possess a) inherent image of an outer world and b) goal to achieve and is thus able to produce a list of actions (plan) to reach the goal. In unfavorable conditions, when the plan is no more applicable, agent is usually able to recompute it.

The process of plan computing (or recomputing) is as follows:

- a sensory input is received by the *belief revision function* and agent's beliefs are altered
- *option generation function* evaluates altered beliefs and intentions and creates the options available to the agent. Agent's desires are constituted.
- *filter function* then considers current beliefs, desires and intentions and produces new intentions
- *action selection function* then receives intentions *filter function* and decides what action to perform

The deliberative agent requires symbolic representation with compositional semantics (e. g. data tree) in all major functions, for its deliberation is not limited to present facts, but construes hypotheses about possible future states and potentially also holds information about past (i.e. memory). These hypothetic states involve goals, plans, partial solutions, hypothetical states of the agent's beliefs, etc. It is evident, that deliberative process may become considerably complex and hardware killing.

History of a concept

Since the early 1970, the *AI planning community* has been involved in developing artificial *planning agent* (a predecessor of a deliberative agent), which would be able to choose a proper plan leading to a specified goal. These early attempts resulted in constructing simple planning system called STRIPS. It soon became obvious that STRIPS concept needed further improvement, for it was unable to effectively solve problems of even moderate complexity. In spite of considerable effort to raise the efficiency (for example by implementing *hierarchical* and *non-linear planning*), the system remained somewhat weak while working with any time-constrained system.

More successful attempts have been made in late 1980s to design *planning agents*. For example the *IPEM* (Integrated Planning, Execution and Monitoring system) had a sophisticated non-linear planner embedded. Further, Wood's *AUTODRIVE* simulated a behavior of deliberative agents in a traffic and Cohen's *PHOENIX* system was construed to simulate a forest fire management.

In 1976, Simon and Newell formulated the Physical Symbol System hypothesis, which claims, that both human and artificial intelligence have the same principle - symbol representation and manipulation. According to the hypothesis it follows, that there is no substantial difference between human and machine in intelligence, but just quantitative and structural - machines are much less complex. Such a provocative proposition must have become the object of serious criticism and raised a wide discussion, but the problem itself still remains unsolved in its merit until these days.

Further development of classical *symbolic AI* proved not to be dependent on final verifying the Physical Symbol System hypothesis at all. In 1988, Bratman, Israel and Pollack introduced *Intelligent Resource-bounded Machine Architecture* (IRMA), the first

system implementing the Belief-Desire-Intention software model (BDI). IRMA exemplifies the standard idea of *deliberative agent* as it is known today: a software agent embedding the symbolic representation and implementing the BDI.

Efficiency of deliberative agents compared to reactive ones

Above-mentioned troubles with symbolic AI have led to serious doubts about the viability of such a concept, which resulted in developing an *reactive architecture*, which is based on wholly different principles. Developers of the new architecture have rejected using symbolic representation and manipulation as a base of any artificial intelligence. Reactive agents achieve their goals simply through reactions on changing environment, which implies reasonable computational modesty.

Even though deliberative agents consume much more system resources than their reactive colleagues, their results are significantly better just in few special situations, whereas it is usually possible to replace one deliberative agent with few reactive ones in many cases, without losing a substantial deal of the simulation result's adequacy. It seems that classical deliberative agents may be usable especially where correct action is required, for their ability to produce optimal, domain-independent solution. Deliberative agent often fails in changing environment, for it is unable to re-plan its actions quickly enough.

Belief-Desire-Intention Software Model

The **Belief-Desire-Intention (BDI) software model** (usually referred to simply, but ambiguously, as **BDI**) is a software model developed for programming intelligent agents. Superficially characterized by the implementation of an agent's *beliefs*, *desires* and *intentions*, it actually uses these concepts to solve a particular problem in agent programming. In essence, it provides a mechanism for separating the activity of selecting a plan (from a plan library) from the execution of currently active plans. Consequently, BDI agents are able to balance the time spent on deliberating about plans (choosing what to do) and executing those plans (doing it). A third activity, creating the plans in the first place (planning), is not within the scope of the model, and is left to the system designer and programmer.

Overview

In order to achieve this separation, the BDI software model implements the principal aspects of Michael Bratman's theory of human practical reasoning (also referred to as Belief-Desire-Intention, or BDI). That is to say, it implements the notions of belief, desire and (in particular) intention, in a manner inspired by Bratman. For Bratman, intention and desire are both pro-attitudes (mental attitudes concerned with action), but intention is distinguished as a conduct-controlling pro-attitude. He identifies commitment as the

distinguishing factor between desire and intention, noting that it leads to (1) temporal persistence in plans and (2) further plans being made on the basis of those to which it is already committed. The BDI software model partially addresses these issues. Temporal persistence, in the sense of explicit reference to time, is not explored. The hierarchical nature of plans is more easily implemented: a plan consists of a number of steps, some of which may invoke other plans. The hierarchical definition of plans itself implies a kind of temporal persistence, since the overarching plan remains in effect while subsidiary plans are being executed.

An important aspect of the BDI software model (in terms of its research relevance) is the existence of logical models through which it is possible to define and reason about BDI agents. Research in this area has led, for example, to the axiomatization of some BDI implementations, as well as to formal logical descriptions such as Anand Rao and Michael Georgeff's BDICTL. The latter combines a multiple-modal logic (with modalities representing beliefs, desires and intentions) with the temporal logic CTL*. More recently, Michael Wooldridge has extended BDICTL to define LORA (the Logic Of Rational Agents), by incorporating an action logic. In principle, LORA allows reasoning not only about individual agents, but also about communication and other interaction in a multi-agent system.

The BDI software model is closely associated with intelligent agents, but does not, of itself, ensure all the characteristics associated with such agents. For example, it allows agents to have private beliefs, but does not force them to be private. It also has nothing to say about agent communication. Ultimately, the BDI software model is an attempt to solve a problem that has more to do with plans and planning (the choice and execution thereof) than it has to do with the programming of intelligent agents.

BDI Agents

A BDI agent is a particular type of bounded rational software agent, imbued with particular *mental attitudes*, viz: Beliefs, Desires and Intentions (BDI).

Architecture

This section defines the idealized architectural components of a BDI system.

- **Beliefs**: Beliefs represent the informational state of the agent, in other words its beliefs about the world (including itself and other agents). Beliefs can also include inference rules, allowing forward chaining to lead to new beliefs. Using the term *belief* rather than *knowledge* recognizes that what an agent believes may not necessarily be true (and in fact may change in the future).
    - **Beliefset**: Beliefs are stored in database (sometimes called a *belief base* or a *belief set*), although that is an implementation decision.
- **Desires**: Desires represent the motivational state of the agent. They represent objectives or situations that the agent *would like* to accomplish or bring about. Examples of desires might be: *find the best price*, *go to the party* or *become rich*.

    - **Goals**: A goal is a desire that has been adopted for active pursuit by the agent. Usage of the term *goals* adds the further restriction that the set of active desires must be consistent. For example, one should not have concurrent goals to go to a party and to stay at home - even though they could both be desirable.
- **Intentions**: Intentions represent the deliberative state of the agent - what the agent *has chosen* to do. Intentions are desires to which the agent has to some extent committed. In implemented systems, this means the agent has begun executing a plan.
    - **Plans**: Plans are sequences of actions (recipes or knowledge areas) that an agent can perform to achieve one or more of its intentions. Plans may include other plans: my plan to go for a drive may include a plan to find my car keys. This reflects that in Bratman's model, plans are initially only partially conceived, with details being filled in as they progress.
- **Events**: These are triggers for reactive activity by the agent. An event may update beliefs, trigger plans or modify goals. Events may be generated externally and received by sensors or integrated systems. Additionally, events may be generated internally to trigger decoupled updates or plans of activity.

BDI Interpreter

This section defines an idealized BDI interpreter that provides the basis of the PRS linage of BDI systems:

1. initialize-state
2. repeat
    1. options: option-generator(event-queue)
    2. selected-options: deliberate(options)
    3. update-intentions(selected-options)
    4. execute()
    5. get-new-external-events()
    6. drop-unsuccessful-attitudes()
    7. drop-impossible-attitudes()
3. end repeat

This basic algorithm has been extended in many ways, for instance to support planning ahead, automated teamwork, and maintenance goals .

Limitations and Criticisms

The BDI software model is one example of a reasoning architecture for a single rational agent, and one concern in a broader multi-agent system. This section bounds the scope of concerns for the BDI software model, highlighting known limitations of the architecture.

- **Learning**: BDI agents lack any specific mechanisms within the architecture to learn from past behavior and adapt to new situations.

- **Three Attitudes**: Classical decision theorists and planning researches question the necessity of having all three attitudes, distributed AI researches question whether the three attitudes are sufficient.
- **Logics**: The multi-modal logics that underlie BDI (that do not have complete axiomatizations and are not efficiently computable) have little relevance in practice.
- **Multiple Agents**: In addition to not explicitly supporting learning, the framework may not be appropriate to learning behavior. Further, the BDI model does not explicitly describe mechanisms for interaction with other agents and integration into a multi-agent system..
- **Explicit Goals**: Most BDI implementations do not have an explicit representation of goals.
- **Lookahead**: The architecture does not have (by design) any lookahead deliberation or forward planning. This may not be desirable because adopted plans may use up limited resources, actions may not be reversible, task execution may take longer than forward planning, and actions may have undesirable side effects if unsuccessful.

BDI Agent Implementations

'Pure' BDI

- Procedural Reasoning System (PRS)
- IRMA (not implemented but can be considered as PRS with non-reconsideration)
- UM-PRS
- Distributed Multi-Agent Reasoning System (dMARS)
- AgentSpeak(L)
- AgentSpeak(RT)
- Agent Real-Time System (ARTS)
- JAM
- JACK Intelligent Agents
- JADEX (Open Source Project)
- Jason (Open Source Project)
- SPARK
- 3APL
- 2APL
- GOAL Agent Programming Language
- CogniTAO (Think-As-One)
- LS/TS - Living Systems Technology Suite

Extensions and Hybrid Systems

- JACK Teams
- CogniTAO (Think-As-One)
- LS/TS - Living Systems Technology Suite
- Brahms

Chapter 7

Contract Net Protocol & Semi Human Instinctive Artificial Intelligence

Contract Net Protocol

Contract Net Protocol (CNP) is a well known task sharing protocol that is used for task allocation in multi-agent systems and consists of a collection of nodes or software agents that form the contract net. Each node on the network can, at different times or for different tasks be a manager or a contractor.

When a node gets a composite task (or for any reason cannot solve the present task) it breaks the problem down into sub-tasks (If possible) and announces the sub-task to the contract net acting as a manager. Bids are then received from potential contractors which the winning contractor(s) are awarded the job(s).

The Contract Net

Task distribution is viewed as a kind of contract negotiation and happens in 5 stages.

1. Recognition
2. Announcement
3. Bidding
4. Awarding
5. Expediting

Recognition

An agent recognises it has a problem that it wants help with. The agent has a goal, and either:

- Realises it cannot achieve the goal in isolation - does not have the capability to fulfil the goal

- Realises it would prefer not to achieve the goal in isolation - typically because of solution quality, deadline, etc.

Announcement

The agent with the task sends out an announcement of the task which includes a specification of the task to be achieved. The specification must encode:

- Description of the task itself
- Any constraints
- Meta-task information

Bidding

Agents that receive the announcement decide themselves whether they should bid for the task. Factors that are taken into consideration are:

- The agent must decide whether it is capable of the expecting task
- The agent must determine the quality constrains and the price information (if relevant)

Awarding

Agents that send the task announcement must choose between the received bids and decide who to award the contract to. The result of this process is communicated to agents that submitted a bid.

Expediting

The successful contractor then expedites the task. This may involve the generation of further contract nets in the form of sub-contracting to complete the task.

Example uses of Contract Net Protocol

An electronic market place for buying and selling goods. For example a system where a user could specify the goods that they want as well as a maximum price that they are willing to pay. The agent programs then would find other user(s) willing to sell the goods within the desired price range. The user with the lowest price would then be selected to fulfill the contract. Other constraints could be applied such as delivery time and the location of the goods.

Semi Human Instinctive Artificial Intelligence

Semi Human Instinctive Artificial Intelligence (**SHIAI**) is a new Artificial Intelligence methodology, first designed to be used in RoboCup competitions. Nowadays it has been used to resolve many different problems.

Overview

The goal of SHIAI is to provide robots (or any other intelligent embedded system) with manlike instincts. SHIAI proposes a nondeterministic decision making theory based on Semi Human Instincts implemented by learned potential fields, using neural networks and fuzzy logic, offline and online learning algorithms, which enable the agent to perform in anonymous, dynamic and non-deterministic environments. SHIAI is like a newly born baby who uses his/her instincts and will gradually become more and more intelligent as the brain learns more about its environment. The use of a new world modeling method called ARPL in SHIAI enables the agent to perform better within anonymous environments where positioning is an important and complex issue.

History

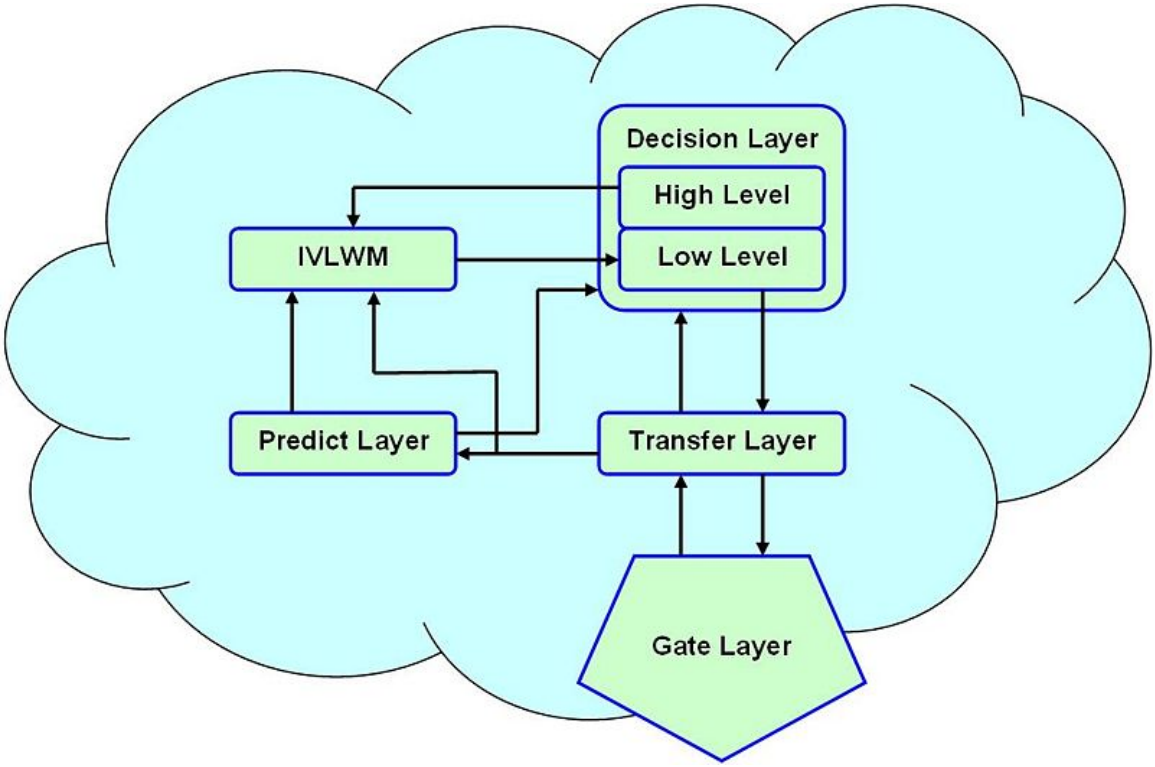
The research and work on this subject started from year 2000 and after 4 years of work and research and consulting with neurologists and psychologists resulted in presenting the MMLAI method. It was practically implemented and tested on the RoboCup Middle Size League (class F-2000) during RoboCup 2004 competitions, which revealed astonishing results some of which not even expected. This achievement encouraged us to work on it harder to cover its weaknesses and make it more optimized and adaptive to perform more efficient in noisy and anonymous environments such as soccer pitch. This led to the invention of SHIAI that was, like MMLAI, practically implemented and tested on the MiddleSize League Robots.

Principles

The first fundamental principle of this theory is based on instinct definition such that every problem has to be partitioned into its main and complex sections and then find a basic but reliable solution for each section using the laws of physics, chemistry, or mathematics. Providing the agent with these collections of instincts, we would have an agent that makes decisions without a particular knowledge and only by its defined instincts even if these decisions are false.
The Second principle is machine learning. In which there are two methods in SHIAI. As a baby learns (meaning both learning with supervisor and without supervisor) and gains experience, he/she would be able to make more optimized decisions and the chosen traveling paths in case of object avoidance will be more accurate.
The third principle of this theory is replacing quantity with quality even within calculations. That is, the volume of calculations is considerable reduced and is more similar to human brain. This will be done using ARPL that has eliminated the need for

exact global positioning. Therefore, relative polar localization substitutes the global positioning where no complex algorithm is required which decreases calculation errors and speeds up the decision making system.
The last principle is decision making under any circumstances. In fact, with this theory we make sure that there is nothing as unexpected condition because basically no conditions are defined in this theory to have unexpected condition.
In SHIAI, depending on the area of performance basic instincts will be defined for the intelligent agent, and then the agent itself nourishes its instincts using learning techniques and special analytical process of the surrounding environment to make more optimized and realistic decisions.

SHIAI Layers



SHIAI Layers

SHI-AI is consisted of five collaborating layers:

Gate Layer (GL)

Gate Layer performs as a gateway between SHI-AI and the surrounding world where all communications between SHIAI and the hardware world are done through this layer. This layer can be compatible with any hardware by making minor changes to the GL structure. Gate Layer is in contact with the Transfer Layer where gathered inputs by the Gate Layer are sent to Transfer Layer and the desired outputs are sent to the Gate Layer by the Transfer Layer.

Transfer Layer (TL)

Transfer Layer is responsible of parsing, correcting, and optimizing all the input and output data. This layer receives inputs from the Gate Layer and, if necessary, will make appropriate changes to the data formats and normalizes them to be ready to be sent to the upper and higher layers. Transfer Layer, also, recognizes errors in input data and will correct them before sending them to the upper or lower layers. This layer synchronously sends the same data that is being sent to the Decision Layer, to IVLWM, Learning and Predict Layers where data will be processed by each of the mentioned layers for different purposes. Finally when the optimum decision has been made by the Decision Layer the output will be sent to the Transfer Layer for optimization and then changed to be ready to be sent to the Gate Layer for final execution.

Decision Layer (DL)

The Decision Layer is consisted of two Low-Level and High-Level sub-layers.

- The Low-Level Decision is based on static laws which are called instinctive decision making in the real world. This decision making method enables the agent to make logical (but not optimized) decisions without a prior learning process, and furthermore provides the agent with unconscious decision making. Unconscious decision making is inevitable in virtual world and specially anonymous and on-deterministic environments. This sub-layer creates the main output of decision layer which is passed to the Transfer Layer to be executed.
- The High Level Decision recognizes and analysis its surrounding environment using appropriate data from the world. The decisions made in this layer are directly influencing the IVLWM. In fact, the decision making process of the agent is to first make a high

level decision having enough information from the world, predicted states, and additional information or commands from other active elements of the environment. This decision is then passed to IVLWM to model a world appropriate for the defined formulas of instincts.

Instinctive Virtual Layered World Modeler (IVLWM)

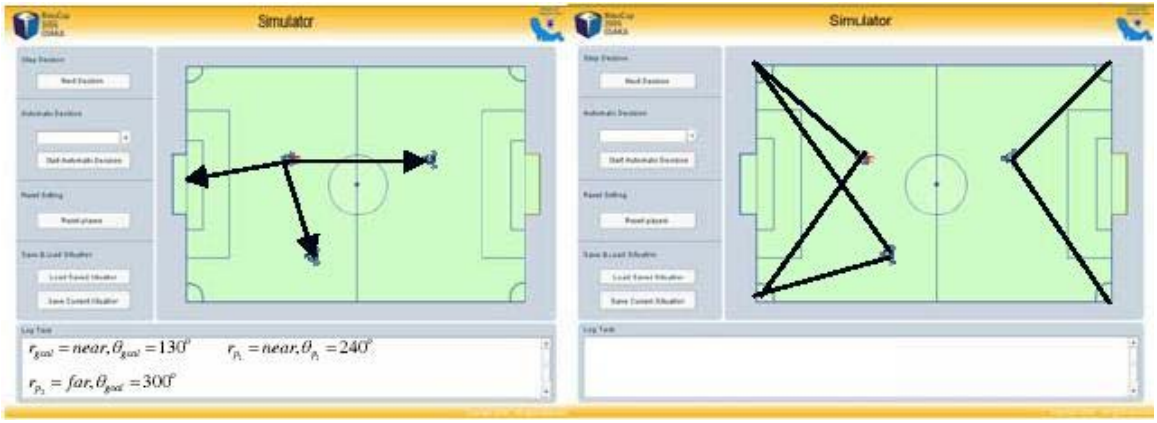
This layer is the most important layer of SHI-AI. As the name implies, IVLWM is responsible for converting the agent's surrounding world to a virtual world where affected by defined laws of instincts formation. The way instincts laws influence the real and virtual worlds depend on decision making conditions and learnings. This layer directly interacts with the learning layer. Thus, IVLWM generates more applicable and optimized virtual world having been fed by the learning process.

Predict Layer (PL)

The Predict Layer is the forecasting side of information processing. The aim here is to derive information about how the surrounding world will be like at some time $t_0 + \varepsilon_t$ in the future, for some ε_t &gt 0, by using data measured up to and including time ε_t. The predicted world is quiet useful for making high level decisions, specially in case of determining action strategies.

The collaboration and communication between layers is done via defined protocols. These protocols have been defined to be compatible with any area of performance by only applying minor changes to the low level making.

Agent Relative Polar Localization (ARPL)



Right picture is similar to the regular positioning techniques and the left picture is a sample of world generation using ARPL technique in SHI-AI Middle-Size Simulator

ARPL is a method for modeling agent's surrounding world based on polar coordinates of r and θ where r represents distance and θ represents angle. In this method, the agent retrieves the location of surrounding objects using the above mentioned coordinate relative to itself. That is, each object will have a distance and angle relative to the agent that results a polar position vector. The collection of these polar position vectors will make the agent's world. To have this method better understood we should now refer to RoboCup implementation of ARPL. In RoboCup implementation of APRL, we may have two ways of expressing the values of distance. The first one would be exact logarithmic value which is actually the logarithmic position of the object in a parabolic mirror, where robots vision is through an omni-directional parabolic mirror (this position is not the exact metric position of the object since it is not re-calculated through the parabolic formula of the mirror). The latter one which is used by Decision Layer is the linguistic fuzzy representation of the distance. This is done by dividing the circular visible area of the robot into several logarithmic sections defined as linguistic quantities like "close", "near", "far", etc. The magnitude of these ranges are increased exponentially from the closest point (tangent point) of the agent to the defined far most point.

Chapter 8

Agent-Based Model in Biology

Agent-based models have many applications in biology, primarily due to the characteristics of the modeling method. Agent-based modeling is a rule-based, computational modeling methodology that focuses on rules and interactions among the individual components or the agents of the system. The goal of this modeling method is to generate populations of the system components of interest and simulate their interactions in a virtual world. Agent-based models start with rules for behavior and seek to reconstruct, through computational instantiation of those behavioral rules, the observed patterns of behavior. Several of the characteristics of agent-based models important to biological studies include:

1. Modular structure: The behavior of an agent-based model is defined by the rules of its agents. Existing agent rules can be modified or new agents can be added without having to modify the entire model.
2. Emergent properties: Through the use of the individual agents that interact locally with rules of behavior, agent-based models result in a synergy that leads to a higher level whole with much more intricate behavior than those of each individual agent.
3. Abstraction: Either by excluding non-essential details or when details are not available, agent-based models can be constructed in the absence of complete knowledge of the system under study. This allows the model to be as simple and verifiable as possible.
4. Stochasticity: Biological systems exhibit behavior that appears to be random. The probability of a particular behavior can be determined for a system as a whole and then be translated into rules for the individual agents.

Before the agent-based model can be developed, one must choose the appropriate software or modeling toolkit to be used. Madey and Nikolai provide an extensive list of toolkits in their paper “Tools of the Trade: A Survey of Various Agent Based Modeling Platforms”. The paper seeks to provide users with a method of choosing a suitable toolkit by examining five characteristics across the spectrum of toolkits: the programming language required to create the model, the required operating system, availability of user

support, the software license type, and the intended toolkit domain. Some of the more commonly used toolkits include Swarm, NetLogo, RePast, and Mason. Listed below are summaries of several articles describing agent-based models that have been employed in biological studies. The summaries will provide a description of the problem space, an overview of the agent-based model and the agents involved, and a brief discussion of the model results.

Forest insect infestations

In the paper titled "Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations", an agent-based model was developed to simulate attack behavior of the Mountain Pine Beetle, *Dendroctonus ponderosae*, (MPB) in order to evaluate how different harvesting policies influence spatial characteristics of the forest and spatial propagation of the MPB infestation over time. About two-thirds of the land in British Columbia, Canada is covered by forests that are constantly being modified by natural disturbances such as fire, disease, and insect infestation. Forest resources make up approximately 15% of the province's economy, so infestations caused by insects such as the MPB can have significant impacts on the economy. The MPB outbreaks are considered a major natural disturbance that can result in widespread mortality of the Lodgepole pine tree, one of the most abundant commercial tree species in British Columbia. Insect outbreaks have resulted in the death of trees over areas of several thousand square kilometers.

The agent-based model developed for this study was designed to simulate the MPB attack behavior in order to evaluate how management practices influence the spatial distribution and patterns of insect population and their preferences for attacked and killed trees. Three management strategies were considered by the model: 1) no management, 2) sanitation harvest and 3) salvage harvest. In the model, the Beetle Agent represented the MPB behavior; the Pine Agent represented the forest environment and tree health evolution; the Forest Management Agent represented the different management strategies. The Beetle Agent follows a series of rules to decide where to fly within the forest and to select a healthy tree to attack, feed, and breed. The MPB typically kills host trees in its natural environment in order to successfully reproduce. The beetle larvae feed on the inner bark of mature host trees, eventually killing them. In order for the beetles to reproduce, the host tree must be sufficiently large and have thick inner bark. The MPB outbreaks end when the food supply decreases to the point that there is not enough to sustain the population or when climatic conditions become unfavorable for the beetle. The Pine Agent simulates the resistance of the host tree, specifically the Lodgepole pine tree, and monitors the state and attributes of each stand of trees. At some point in the MPB attack, the number of beetles per tree reaches the host tree capacity. When this point is reached, the beetles release a chemical to direct beetles to attack other trees. The Pine Agent models this behavior by calculating the beetle population density per stand and passes the information to the Beetle Agents. The Forest Management Agent was used, at the stand level, to simulate two common silviculture practices (sanitation and salvage) as well as the strategy where no management practice was employed. With the sanitation harvest strategy, if a stand has an infestation rate greater than a set threshold,

the stand is removed as well as any healthy neighbor stand when the average size of the trees exceeded a set threshold. For the salvage harvest strategy, a stand is removed even it is not under a MPB attack if a predetermined number of neighboring stands are under a MPB attack.

The study considered a forested area in the North-Central Interior of British Columbia of approximately 560 hectare. The area consisted primarily of Lodgepole pine with smaller proportions of Douglas fir and White spruce. The model was executed for five time steps, each step representing a single year. Thirty simulation runs were conducted for each forest management strategy considered. The results of the simulation showed that when no management strategy was employed, the highest overall MPB infestation occurred. The results also showed that the salvage forest management technique resulted in a 25% reduction in the number of forest strands killed by the MPB, as opposed to a 19% reduction by the salvage forest management strategy. In summary, the results show that the model can be used as a tool to build forest management policies.

Invasive species

Invasive species refers to "non-native" plants and animals that adversely affect the environments they invade. The introduction of invasive species may have environmental, economic, and ecological implications. In the paper titled "An Agent-Based Model of Border Enforcement for Invasive Species Management", an agent-based model is presented that was developed to evaluate the impacts of port-specific and importer-specific enforcement regimes for a given agricultural commodity that presents invasive species risk. Ultimately, the goal of the study was to improve the allocation of enforcement resources and to provide a tool to policy makers to answer further questions concerning border enforcement and invasive species risk.

The agent-based model developed for the study considered three types of agents: invasive species, importers, and border enforcement agents. In the model, the invasive species can only react to their surroundings, while the importers and border enforcement agents are able to make their own decisions based on their own goals and objectives. The invasive species has the ability to determine if it has been released in an area containing the target crop, and to spread to adjacent plots of the target crop. The model incorporates spatial probability maps that are used to determine if an invasive species becomes established. The study focused on shipments of broccoli from Mexico into California through the ports of entry Calexico, California and Otay Mesa, California. The selected invasive species of concern was the crucifer flea beetle (*Phyllotreta cruciferae*). California is by far the largest producer of broccoli in the United States and so the concern and potential impact of an invasive species introduction through the chosen ports of entry is significant. The model also incorporated a spatially explicit damage function that was used to model invasive species damage in a realistic manner. Agent-based modeling provides the ability to analyze the behavior of heterogeneous actors, so three different types of importers were considered that differed in terms of commodity infection rates (high, medium, and low), pretreatment choice, and cost of transportation to the ports. The model gave predictions on inspection rates for each port of entry and importer and determined the

success rate of border agent inspection, not only for each port and importer but also for each potential level of pretreatment (no pretreatment, level one, level two, and level three).

The model was implemented and ran in NetLogo, version 3.1.5. Spatial information on the location of the ports of entry, major highways, and transportation routes was included in the analysis as well as a map of California broccoli crops layered with invasive species establishment probability maps. BehaviorSpace, a software tool integrated with NetLogo, was used to test the effects of different parameters (e.g. shipment value, pretreatment cost) in the model. On average, 100 iterations were calculated at each level of the parameter being used, where an iteration represented a one-year run.

The results of the model showed that as inspection efforts increase, importers increase due care, or the pretreatment of shipments, and the total monetary loss of California crops decreases. The model showed that importers respond differently to an increase in inspection effort. Some importers responded to increased inspection rate by increasing pretreatment effort, while others chose to avoid shipping to a specific port, or shopped for another port. An important result of the model results is that it can show or provide recommendations to policy makers about the point at which importers may start to shop for ports, such as the inspection rate at which port shopping is introduced and the importers associated with a certain level of pest risk or transportation cost are likely to make these changes. Another interesting outcome of the model is that when inspectors were not able to learn to respond to an importer with previously infested shipments, damage to California broccoli crops was estimated to be $150 million. However, when inspectors were able to increase inspection rates of importers with previous violations, damage to the California broccoli crops was reduced by approximately 12%. The model provides a mechanism to predict the introduction of invasive species from agricultural imports and their likely damage. Equally as important, the model provides policy makers and border control agencies with a tool that can be used to determine the best allocation of inspectional resources.

Aphid population dynamics

In the article titled “Aphid Population Dynamics in Agricultural Landscapes: An Agent-based Simulation Model”, an agent-based model is presented to study the population dynamics of the bird cherry-oat aphid, *Rhopalosiphum padi* (L.). The study was conducted in a five square kilometer region of North Yorkshire, a county located in the Yorkshire and the Humber region of England. The agent-based modeling method was chosen because of its focus on the behavior of the individual agents rather than the population as a whole. The authors propose that traditional models that focus on populations as a whole do not take into account the complexity of the concurrent interactions in ecosystems, such as reproduction and competition for resources which may have significant impacts on population trends. The agent-based modeling approach also allows modelers to create more generic and modular models that are more flexible and easier to maintain than modeling approaches that focus on the population as a whole. Other proposed advantages of agent-based models include realistic representation of a

phenomenon of interest due to the interactions of a group of autonomous agents, and the capability to integrate quantitative variables, differential equations, and rule based behavior into the same model.

The model was implemented in the modeling toolkit Repast using the JAVA programming language. The model was ran in daily time steps and focused on the autumn and winter seasons. Input data for the model included habitat data, daily minimum, maximum, and mean temperatures, and wind speed and direction. For the Aphid agents, age, position, and morphology (alate or apterous) were considered. Age ranged from 0.00 to 2.00, with 1.00 being the point at which the agent becomes an adult. Reproduction by the Aphid agents is dependent on age, morphology, and daily minimum, maximum, and mean temperatures. Once nymphs hatch, they remain in the same location as their parents. The morphology of the nymphs is related to population density and the nutrient quality of the aphid's food source. The model also considered mortality among the Aphid agents, which is dependent on age, temperatures, and quality of habitat. The speed at which an Aphid agent ages is determined by the daily minimum, maximum, and mean temperatures. The model considered movement of the Aphid agents to occur in two separate phases, a migratory phase and a foraging phase, both of which affect the overall population distribution.

The study started the simulation run with an initial population of 10,000 alate aphids distributed across a grid of 25 meter cells. The simulation results showed that there were two major population peaks, the first in early autumn due to an influx of alate immigrants and the second due to lower temperatures later in the year and a lack of immigrants. Ultimately, it is the goal of the researchers to adapt this model to simulate broader ecosystems and animal types.

Aquatic population dynamics

In the article titled "Exploring Multi-Agent Systems In Aquatic Population Dynamics Modeling", a model is proposed to study the population dynamics of two species of macrophytes. Aquatic plants play a vital role in the ecosystems in which they live as they may provide shelter and food for other aquatic organisms. However, they may also have harmful impacts such as the excessive growth of non-native plants or eutrophication of the lakes in which they live leading to anoxic conditions. Given these possibilities, it is important to understand how the environment and other organisms affect the growth of these aquatic plants to allow mitigation or prevention of these harmful impacts.

Potamogeton pectinatus is one of the aquatic plant agents in the model. It is an annual growth plant that absorbs nutrients from the soil and reproduces through root tubers and rhizomes. Reproduction of the plant is not impacted by water flow, but can be influenced by animals, other plants, and humans. The plant can grow up to two meters tall, which is a limiting condition because it can only grow in certain water depths, and most of its biomass is found at the top of the plant in order to capture the most sunlight possible. The second plant agent in the model is *Chara aspera*, also a rooted aquatic plant. One major difference in the two plants is that the latter reproduces through the use of very small

seeds called oospores and bulbills which are spread via the flow of water. *Chara aspera* only grows up to 20 cm and requires very good light conditions as well as good water quality, all of which are limiting factors on the growth of the plant. *Chara aspera* has a higher growth rate than *Potamogeton pectinatus* but has a much shorter life span. The model also considered environmental and animal agents. Environmental agents considered included water flow, light penetration, and water depth. Flow conditions, although not of high importance to *Potamogeton pectinatus*, directly impact the seed dispersal of *Chara aspera*. Flow conditions affect the direction as well as the distance the seeds will be distributed. Light penetration strongly influences *Chara aspera* as it requires high water quality. Extinction coefficient (EC) is a measure of light penetration in water. As EC increases, the growth rate of *Chara aspera* decreases. Finally, depth is important to both species of plants. As water depth increases, the light penetration decreases making it difficult for either species to survive beyond certain depths.

The area of interest in the model was a lake in the Netherlands named Lake Veluwe. It is a relatively shallow lake with an average depth of 1.55 meters and covers about 30 square kilometers. The lake is under eutrophication stress which means that nutrients are not a limiting factor for either of the plant agents in the model. The initial position of the plant agents in the model was randomly determined. The model was implemented using Repast software package and was executed to simulate the growth and decay of the two different plant agents, taking into account the environmental agents previously discussed as well as interactions with other plant agents. The results of the model execution show that the population distribution of *Chara aspera* has a spatial pattern very similar to the GIS maps of observed distributions. The authors of the study conclude that the agent rules developed in the study are reasonable to simulate the spatial pattern of macrophyte growth in this particular lake.

Chapter 9

Osmius

Osmius

Osmius Desktop

Developer(s)	Peopleware SL
Initial release	July 14, 2006
Stable release	10.7 / July 23, 2010; 8 months ago
Development status	Active
Written in	C++, Java, Groovy
Operating system	Linux, Windows, Solaris
Available in	English, Spanish, French
Type	System monitoring, Business monitoring, KPI tracking

Osmius is a monitoring tool prepared to deal with thousands of devices, servers, applications and business indicators with incredible performance. Osmius was designed from the beginning to offer both the technical and the business view of your systems so the user can see that Server "Moon" is down but also what Services and Applications are affected. Osmius keeps track of the state and availability of every component and check

the Service Level Agreements regularly providing Business Intelligence and DataMining tools and Control Panels.

Overview

These are, in short, Osmius main features:

- **Prepared to monitor Everything**: In addition to CPU Load and Network traffic, why don't you monitor the *number of sells per hour*? Or the time to upload a document to your CMS?
- **SLA management**. Services, dependencies, propagation rules, capacity planning and ITIL best practices.
- **Performance**: More than 60.000 events per minute. Thousands of devices (things) and millions of events.
- **Multiplatform**: From Unix to Windows, Databases from MySQL to Oracle, Virtual Machine Engines, etc. Easy to develop new agents.
- **Customize it!** Use and integrate your own technical team scripts, add new SNPM or WMI events. Get new plugins from the Osmius portal or, better, make your owns.
- **Integrated GIS**. Locate your instances and leverage on the GIS engine capabilities.
- **Notification system**. Want to receive alerts into your mobile? Send alerts via e-mail, jabber or SMS to the 24×7 support team?
- **AutoDiscovery**: Discover the elements in your network by type: Databases, Webs, Servers, SNMP devices.
- **Customized Desktop**: Based on Widgets.
- **Mobile Console**: Check from iPhone or Android devices the real time state or the SLA Dash board.

- **Professional Support**: Peopleware is the company behind Osmius providing continuous improvements, bug fixing and development and support services.

History

Osmius started as an idea from some of the final development team members to address the problem of 2000's monitoring tools:

- Hard to implement and difficult to understand.
- Privative tools were very expensive and closed to learn and to expand.
- Free Tools were too technical, complicated and based on heavy scripting.
- Documentation is expensive or incomplete (or non-existent).
- Few of them were Business Oriented and ITIL integrated tools.

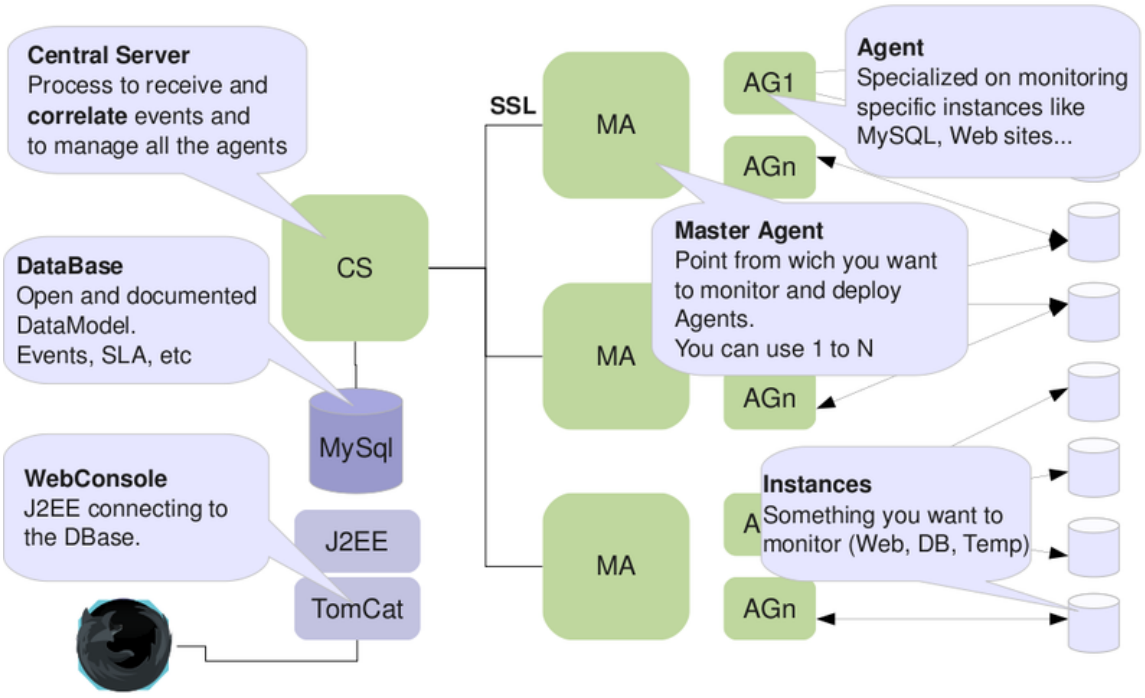
After spending years in the monitoring career, it was clear that there is room and that there are needs about monitoring systems and services, and that an initially small team could design a new tool with these traits:

- Easy to understand. Few and powerful concepts.
- Easy to administer. No scattered scripting and no more lack of central management.
- Robust and scalable. Real time event processing and tiny footprints on the monitored "things".
- Business Oriented. SLA management, offering technical impact analysis as well as business views.
- Prepared to monitor KPIs whether they are CPU Load or number of sells in Michigan.
- Open: Open Source, open documentation.
- Professional: Support, development and consulting services.

In 2010 Osmius is a consolidated product with thousands of download from SourceForge.

Architecture

Osmius is a web based interface tool from which the user can manage the overall infrastructure that can be as simple as a single Central Osmius server, or use Master Agents from where monitor and administer monitoring and monitored platform.



You can choose between intrusive (agent based) or external monitoring depending on your needs and restrictions. If any the connections with the agents are properly encrypted and there are process in the Central Server that deal with tasks such as sending new configuration parameters, prepare a server to monitor new systems or applications or to receive measurements in a robust and reliable way.

Osmius uses MySQL and the underlying storage engine (InnoDB) and is prepared to receive millions of events every day. Osmius implements a Round Robin Database policy so the user can apply event grouping as the data gets older and loose relevance.

New monitoring capabilities can be added creating user defined events or adding plug-ins to the platform, and there are modules to create new events based on current measures and to derive new data using statistical functions.

The Osmius Engine (agents, master agen and, central processes) is based on ACE ADAPTIVE Communication Environment and thanks to that we have multiplatform capabilities (code once, compile many) and real time as well as very small footprints.

Monitoring Capabilities

Osmius agents can monitor devices, systems and applications in Windows, Linux, Solaris and HP-UX and they normally rely on native APIs and connection pools to reduce footprints. Every **Osmius agent** can be configured to use "silent mode" to prevent network resources starvation and they are intelligent enough to recover lost from lost connections and even from process failures.

Operative systems

Linux: local and remote monitoring. Monitors uptime, system resource consumption, processes running state, resource consumption per process, filesystems, system temperature, network traffic.

Solaris: local monitoring. Monitors uptime, system resource consumption (cpu, memory, swap ...), processes and daemons running state, filesystems, users connected ...

Windows: local and remote monitoring. Monitors uptime, system resources consumption (cpu, memory, disk, virtual memory ...), processes and services running state, Windows EventLog, network traffic, resources consumption per process, your own WMI queries

Databases

MS SQL Server: local and remote monitoring. Monitors memory used by database server, splitted pages, used space by log and data, locks, number of connections, free pages, batch request ...

MySQL: local and remote monitoring. Monitors connections, threads, network traffic, queries per second, locks, slow queries, uptime, open tables ...

Oracle: local and remote monitoring. Monitors availability, uptime, locks, connections, invalid objects, performance problems, extensions problems, used space, slow queries, tablespaces usage, data and instructions cache success percentage, cursor consumption ...

PostgreSQL: local and remote monitoring. Monitors availability, number of connections, percentage of maximum connections used and database size in Megabytes.

SNMP Devices

Osmius agent to obtain v1 and v2c SNMP **variables** and **traps** from **any SNMP device**.

In addition Osmius have a preconfigured agent to monitors **generic CISCO routers**.Osmius has **Autodiscovery** of new traps.

Applications

Apache: local and remote monitoring. Monitors availability, uptime, request and a lot of data from your Apache server.

Asterisk: local and remote monitoring. Using AMI monitors availability, uptime, extensions state and mailbox messages.

Exchange: local and remote monitoring. Monitors Exchange 2003 and 2007 performance counters such as connected users, queues size, delivered, sent and failed messages, RPC requests and latency.

IIS: local and remote monitoring. Monitors uptime, connections, network traffic, request types and IIS errors.

Tomcat: local and remote monitoring. Using JMX monitors availability, uptime, applications running state, deployment status, memory usage, threads, pending objects and processor usage.

Technologies

SSH: local and remote monitoring. Monitors any SSH able device and use your own commands.

WMI: local and remote monitoring. Monitors any Microsoft and Windows WMI application.

IP: ping time reply, service availability such us http, ftp, ssh, telnet, imap, pop3, smtp ... at any IP address.

IPMI: local and remote monitoring. Monitors availability, uptime and any data from IPMI able device sensors.

LOG: local monitoring. Monitors log file size and content using regular expressions.

WEB: local and remote monitoring. Monitors ,with extraordinary performance, web availability, load time, transfer time, download size and content (using regular expressions) of any http or https source.

Check for updates about Osmius monitoring capabilities here.

Performance

In order to guarantee Osmius scalability and its smooth running in data intensive installations Osmius is tested in high-performance-requirement environments. Over an Intel Core Duo 2.5 GHz:

- Process 1.500 events each minute.
- Store millions of measures.
- Monitor thousands of instances.
- Manage the SLAs of thousands of services.
- Desploy 1000 agents in 15 minutes aprox.

Check about Osmius requirements and benchmarking results here

Releases

Osmius releases are numbered the same way as Ubuntu does. That is two figures for the year and the next two for the month.

Osmius 10.07 corresponds to July, 2010 release.

The Osmius development Team create a new version every month, but they are only made public twice a year, normally in January and July every year.

Chapter 10

Pandora FMS

Pandora FMS

Pandora FMS v 3.0 Dashboard

Original author(s)	Sancho Lerena
Developer(s)	Ártica ST
Initial release	October 14, 2004
Stable release	3.2.1 / February 23, 2011; 35 days ago
Development status	Active
Written in	Perl, PHP
Operating system	Linux
Available in	English, Spanish, Italian, Polish
Type	Network monitoring
License	GNU General Public License or proprietary EULA

Pandora FMS (for *Pandora Flexible Monitoring System*) is software solution for monitoring computer networks. Pandora FMS allows monitoring in a visual way the status and performance of several parameters from different operating systems, servers, applications and hardware systems such as firewalls, proxies, databases, web servers or routers.

Pandora FMS can be deployed in almost any operating system. It features remote monitoring (WMI, SNMP, TCP. UDP, ICMP, HTTP...) and it can also use agents. An agent is available for each platform. It can also monitor hardware systems with a TCP/IP stack, such as load balancers, routers, network switches, printers or firewalls.

Pandora FMS has several servers that process and get information from different sources, using WMI for gathering remote Windows information, a predictive server, a plug-in server which makes complex user-defined network tests, an advanced export server to replicate data between different sites of Pandora FMS, a network discovery server, and an SNMP Trap console.

Released under the terms of the GNU General Public License, Pandora FMS is free software.

Components

Pandora Servers and SNMP Console

In Pandora FMS architecture, servers are the core of the system because they are the recipients of bundles of information. They also generate monitoring alerts. It is possible to have different modular configurations for the servers: several servers for very big systems, or just a single server. Servers are also responsible for inserting the gathered data into Pandora's database. It is possible to have several Pandora Servers connected to a single Database.

Servers are developed in Perl and work on any platform that has the required modules. Pandora was originally developed for Linux.

Web console

Pandora's user interface allows people to operate and manage the monitoring system. It is developed in PHP and depends on a database and a web server. It can work in in a wide range of platforms: Linux, Solaris, Microsoft Windows, AIX and others. Several web consoles can be deployed in the same system if required.

Agents

Agents are daemons or services that can monitor any numeric parameter, boolean status, string or numerical incremental data and/or condition. They can be developed in any language (as Shellscript, WSH, Perl or C). They run on any type of platform (Microsoft,

AIX, Solaris, Linux, IPSO, Mac OS or FreeBSD), also SAP, because the agents can communicate with the Pandora FMS Servers to send data in XML using SSH, FTP, NFS, Tentacle (protocol) or any data transfer mean.

Database

The database module is the core module of Pandora. All the information of the system resides here. For example, all data gathered by agents, configuration defined by administrator, events, incidents, audit info, etc. are stored in the database.

At present, only MySQL database is supported, although support for others is planned to be added in the future.

Releases

Stable releases

Pandora FMS has been stable since version 1.0.

Version	Date	Information
3.2	December 27, 2010	Performance improved, smartphone web console, topology maps improved, policy manager improved, new file distribution for agents and new languages supported in reports.
3.1	June 01, 2010	
3.0	December 29, 2009	Flash interactive graphs, alert improvement and redesign, important improvements on windows agent, new SNMP trap console
2.1	March 3, 2009	Multi server support, Tentacle server and client included in packages, usability and performance improvements
2.0	September 3, 2008	New WMI, Plugin and Prediction Server. Better network auto discovery and topology detection, correlated alerts, log parser capable agents (win, Unix), planned downtimes, better reporting and more.
1.3.1	April 30, 2008	New transfer protocol: Tentacle (protocol), new server options
1.3	October 12, 2007	New logo, new recon server
1.2	December 5, 2006	New logo, renamed as Pandora FMS, changed homepage
1.1	May 13, 2005	New agents, console and documentation

1.0	October 14, 2004	First stable release (known as pandoramon)

Software Appliances

With the release of Pandora FMS 3.0 the pandora team had made available several software appliances of Pandora FMS.

VMWare or VirtualBox Image

Pandora Team created an OpenSUSE 11.1 VMware test image with final version of Pandora FMS 3.0. This image could be used with VMWare Player, VMWare Server or VirtualBox.

It has all the Pandora FMS components installed and MySQL, Apache2 and PHP with all modules dependencies resolved, including Pear Graph for reporting.

Chapter 11

Procedural Reasoning System

In Artificial Intelligence, the **Procedural Reasoning System (PRS)** is a framework for constructing real-time reasoning systems that can perform complex tasks in dynamic environments. It is based on the notion of a Rational agent or Intelligent agent using the Belief-Desire-Intention software model.

Overview

A user application is predominately defined, and provided to a PRS system is a set of *knowledge areas*. Each knowledge area is a piece of procedural knowledge that specifies how to do something, e.g., how to navigate down a corridor, or how to plan a path (in contrast with robotic architectures where the programmer just provides a model of what the states of the world are and how the agent's primitive actions affect them). Such a program, together with the PRS interpreter, is used to control the agent.

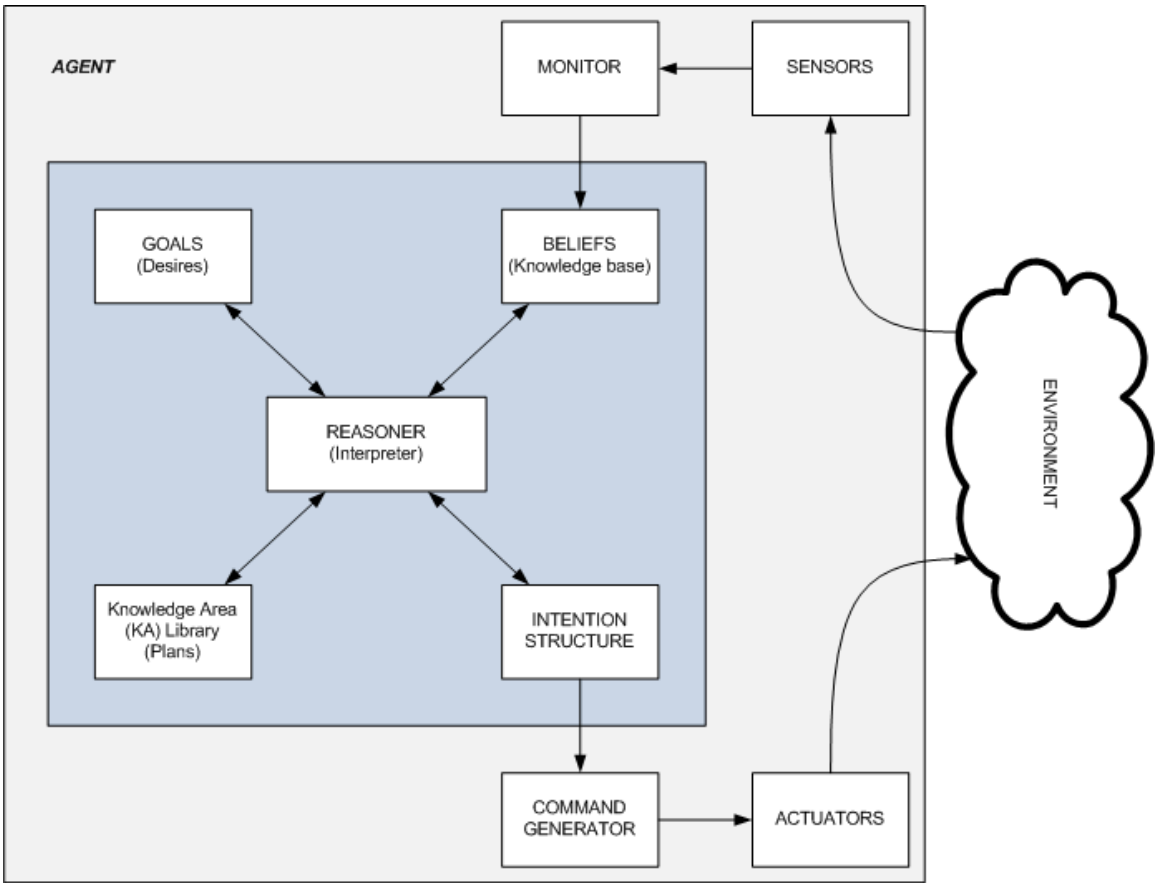
The interpreter is responsible for maintaining beliefs about the world state, choosing which goals to attempt to achieve next, and choosing which knowledge area to apply in the current situation. How exactly these operations are performed might depend on domain-specific meta-level knowledge areas. Unlike traditional AI planning systems that generate a complete plan at the beginning, and replan if unexpected things happen, PRS interleaves planning and doing actions in the world. At any point, the system might only have a partially specified plan for the future.

PRS is based on the BDI or belief-desire-intention framework for intelligent agents. Beliefs consist of what the agent believes to be true about the current state of the world, desires consist of the agent's goals, and intentions consist of the agent's current plans for achieving those goals. Furthermore, each of these three components is typically *explicitly* represented somewhere within the memory of the PRS agent at runtime, which is in contrast to purely reactive systems, such as the subsumption architecture.

History

The PRS was developed by the Artificial Intelligence Center at SRI International during the 1980s. Many were involved in the development of the PRS, not limited to Michael Georgeff, Amy L. Lansky, and François Félix Ingrand. The framework was responsible for exploiting and popularizing the Belief-Desire-Intention model in software for control of an Intelligent agent. The seminal application of the framework was a fault detection system for the Reaction Control System of the NASA Space Shuttle Discovery. Development on the PRS continued at the Australian Artificial Intelligence Institute through to the late 1990s which lead to the development of a C++ implementation and extension called dMARS.

Architecture



Depiction of the PRS Architecture.

The system architecture of PRS is composed of the following components:

- **Database** for beliefs about the world, represented using first order predicate calculus.
- **Goals** to be realized by the system as conditions over an interval of time on internal and external state descriptions (beliefs).

- **Knowledge Areas** (KA) or plans that define sequences of low-level actions toward achieving a goal in specific situations.
- **Intentions** that include those KA that have been selection for current and eventual execution.
- **Interpreter** or inference mechanism that manages the system.

Features

The PRS was developed for embedded application in dynamic and real-time environments. As such it specifically addressed the limitations of other contemporary control and reasoning architectures like Expert Systems and the Blackboard system. The following define the general requirements for the development of the PRS:

- asynchronous event handling
- guaranteed reaction and response types
- procedural representation of knowledge
- handling of multiple problems
- reactive and goal-directed behavior
- focus of attention
- reflective reasoning capabilities
- continuous embedded operation
- handling of incomplete of inaccurate data
- handling of transients
- modeling delayed feedback
- operator control

Applications

The seminal application of the PRS was a monitoring and fault detection system for the Reaction Control System (RCS) on the NASA space shuttle. The RCS provides propulsive forces from a collection of jet thrusters and controls attitude of the space shuttle. A PRS-based fault diagnostic system was developed and tested using a simulator. It included over 100 KAs and over 25 meta level KAs. RCS specific KAs were written by space shuttle mission controllers. It was implemented on the Symbolics 3600 Series LISP machine and used multiple communicating instances of PRS. The system maintained over 1000 facts about the RCS, over 650 facts for the forward RCS alone and half of which are updated continuously during the mission. A version of the PRS was used to monitor the Reaction Control System on the NASA Space Shuttle Discovery.

PRS was tested on Shakey the robot including navigational and simulated jet malfunction scenarios based on the space shuttle. Later applications included a network management monitor called the Interactive Real-time Telecommunications Network Management System (IRTNMS) for Telecom Australia.

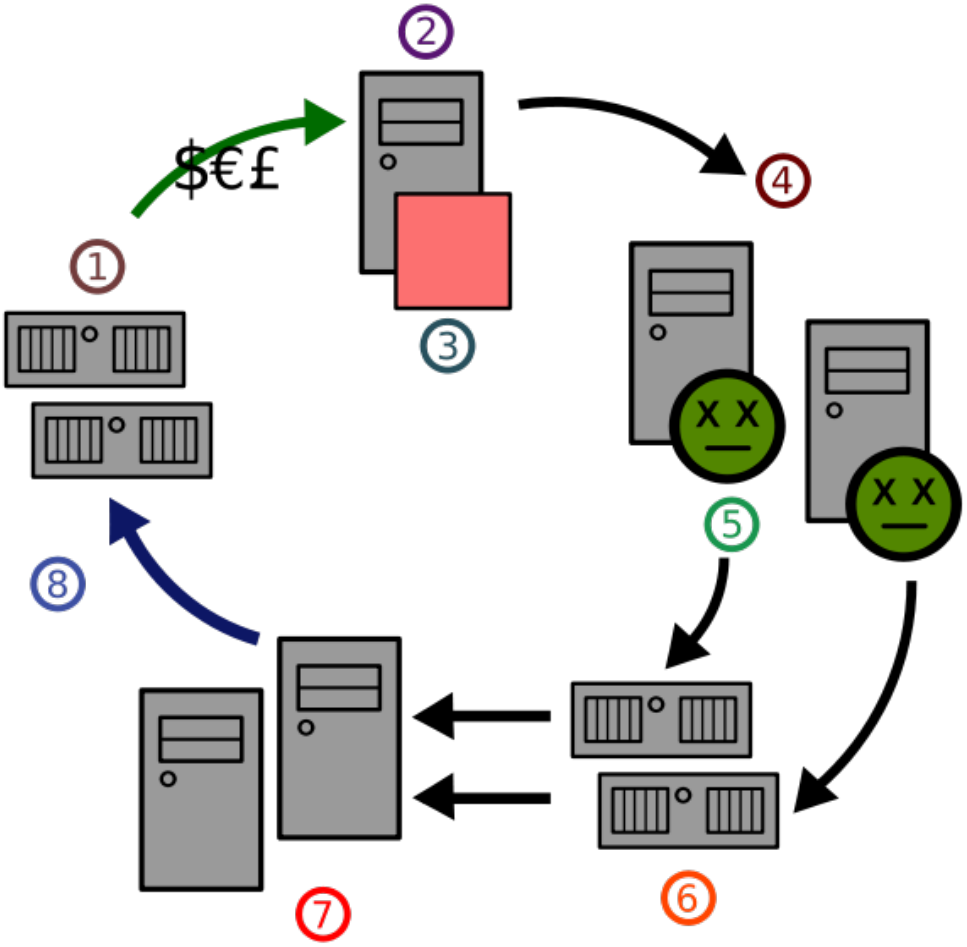
Extensions

The following list the major implementations and extensions of the PRS architecture.

- UM-PRS
- OpenPRS (formerly C-PRS and Propice)
- AgentSpeak
- Distributed Multi-Agent Reasoning System (dMARS)
- JAM
- JACK Intelligent Agents
- SRI Procedural Agent Realization Kit (SPARK)
- PRS-CL

Chapter 12

Storm Botnet



The typical lifecycle of spam that originates from a botnet:
(1) Spammer's web site (2) Spammer (3) Spamware (4) Infected computers (5) Virus or trojan (6) Mail servers (7) Users (8) Web traffic

The **Storm botnet** or **Storm worm botnet** (not to be confused with StormBot, which is a TCL script that is not malicious) is a remotely controlled network of "zombie" computers (or "botnet") that has been linked by the Storm Worm, a Trojan horse spread through e-mail spam. Some have estimated that by September 2007 the Storm botnet was running on anywhere from 1 million to 50 million computer systems. Other sources have placed the size of the botnet to be around 250,000 to 1 million compromised systems. More conservatively, one network security analyst claims to have developed software that has crawled the botnet and estimates that it controls 160,000 infected computers. The Storm botnet was first identified around January 2007, with the Storm worm at one point accounting for 8% of all malware on Microsoft Windows computers.

The Storm botnet has been used in a variety of criminal activities. Its controllers and the authors of the Storm Worm have not yet been identified. The Storm botnet has displayed defensive behaviors that indicated that its controllers were actively protecting the botnet against attempts at tracking and disabling it. The botnet has specifically attacked the online operations of some security vendors and researchers who attempted to investigate the botnet. Security expert Joe Stewart revealed that in late 2007, the operators of the botnet began to further decentralize their operations, in possible plans to sell portions of the Storm botnet to other operators. Some reports as of late 2007 indicated the Storm botnet to be in decline, but many security experts reported that they expect the botnet to remain a major security risk online, and the United States Federal Bureau of Investigation considers the botnet a major risk to increased bank fraud, identity theft, and other cybercrimes.

The botnet is reportedly powerful enough as of September 2007 to force entire countries off the Internet, and is estimated to be capable of executing more instructions per second than some of the world's top supercomputers. However, it is not a completely accurate comparison, according to security analyst James Turner, who said that comparing a botnet to a supercomputer is like comparing an army of snipers to a nuclear weapon. Bradley Anstis, of the United Kingdom security firm Marshal, said, "The more worrying thing is bandwidth. Just calculate four million times a standard ADSL connection. That's a lot of bandwidth. It's quite worrying. Having resources like that at their disposal—distributed around the world with a high presence and in a lot of countries—means they can deliver very effective distributed attacks against hosts."

Origins

First detected on the Internet in January 2007, the Storm botnet and worm are so-called because of the storm-related subject lines its infectious e-mail employed initially, such as "230 dead as storm batters Europe." Later provocative subjects included, "Chinese missile shot down USA aircraft," and "U.S. Secretary of State Condoleezza Rice has kicked German Chancellor Angela Merkel." It is suspected by some information security professionals that well-known fugitive spammers, including Leo Kuvayev, may be involved in the operation and control of the Storm botnet. According to technology journalist Daniel Tynan, writing under his "Robert X. Cringely" pseudonym, a great portion of the fault for the existence of the Storm botnet lay with Microsoft and Adobe

Systems. Other sources state that Storm Worm's primary method of victim acquisition is through enticing users via frequently changing social engineering (confidence trickery) schemes. According to Patrick Runald, the Storm botnet has a strong American focus, and likely has agents working to support it within the United States. Some experts, however, believe the Storm botnet controllers are Russian, some pointing specifically at the Russian Business Network, citing that the Storm software mentions a hatred of the Moscow-based security firm Kaspersky Lab, and includes the Russian word *"buldozhka,"* which means "bulldog."

Composition

The botnet, or zombie network, comprises computers running Microsoft Windows as their operating system. Once infected, a computer becomes known as a bot. This bot then performs automated tasks—anything from gathering data on the user, to attacking web sites, to forwarding infected e-mail—without its owner's knowledge or permission. Estimates indicate that 5,000 to 6,000 computers are dedicated to propagating the spread of the worm through the use of e-mails with infected attachments; 1.2 billion virus messages have been sent by the botnet through September 2007, including a record 57 million on August 22, 2007 alone. Lawrence Baldwin, a computer forensics specialist, was quoted as saying, "Cumulatively, Storm is sending billions of messages a day. It could be double digits in the billions, easily." One of the methods used to entice victims to infection-hosting web sites are offers of free music, for artists such as Beyoncé Knowles, Kelly Clarkson, Rihanna, The Eagles, Foo Fighters, R. Kelly, and Velvet Revolver. Signature-based detection, the main defense of most computer systems against virus and malware infections, is hampered by the large number of Storm variants.

Back-end servers that control the spread of the botnet and Storm worm automatically re-encode their distributed infection software twice an hour, for new transmissions, making it difficult for anti-virus vendors to stop the virus and infection spread. Additionally, the location of the remote servers which control the botnet are hidden behind a constantly changing DNS technique called ‘fast flux’, making it difficult to find and stop virus hosting sites and mail servers. In short, the name and location of such machines are frequently changed and rotated, often on a minute by minute basis. The Storm botnet's operators control the system via peer-to-peer techniques, making external monitoring and disabling of the system more difficult. There is no central "command-and-control point" in the Storm botnet that can be shut down. The botnet also makes use of encrypted traffic. Efforts to infect computers usually revolve around convincing people to download e-mail attachments which contain the virus through subtle manipulation. In one instance, the botnet's controllers took advantage of the National Football League's opening weekend, sending out mail offering "football tracking programs" which did nothing more than infect a user's computer. According to Matt Sergeant, chief anti-spam technologist at MessageLabs, "In terms of power, [the botnet] utterly blows the supercomputers away. If you add up all 500 of the top supercomputers, it blows them all away with just 2 million of its machines. It's very frightening that criminals have access to that much computing power, but there's not much we can do about it." It is estimated that only 10%-20% of the total capacity and power of the Storm botnet is currently being used.

Computer security expert Joe Stewart detailed the process by which compromised machines join the botnet: attempts to join the botnet are made by launching a series of EXE files on the said machine, in stages. Usually, they are named in a sequence from *game0.exe* through *game5.exe*, or similar. It will then continue launching executables in turn. They typically perform the following:

1. `game0.exe` - Backdoor/downloader
2. `game1.exe` - SMTP relay
3. `game2.exe` - E-mail address stealer
4. `game3.exe` - E-mail virus spreader
5. `game4.exe` - Distributed denial of service (DDoS) attack tool
6. `game5.exe` - Updated copy of Storm Worm dropper

At each stage the compromised system will connect into the botnet; fast flux DNS makes tracking this process exceptionally difficult. This code is run from *%windir%\system32\wincom32.sys* on a Windows system, via a kernel rootkit, and all connections back to the botnet are sent through a modified version of the eDonkey/Overnet communications protocol.

Methodology

The Storm botnet and its variants employ a variety of attack vectors, and a variety of defensive steps exist as well. The Storm botnet was observed to be defending itself, and attacking computer systems that scanned for Storm virus-infected computer systems online. The botnet will defend itself with DDoS counter-attacks, to maintain its own internal integrity. At certain points in time, the Storm worm used to spread the botnet has attempted to release hundreds or thousands of versions of itself onto the Internet, in a concentrated attempt to overwhelm the defenses of anti-virus and malware security firms. According to Joshua Corman, an IBM security researcher, "This is the first time that I can remember ever seeing researchers who were actually afraid of investigating an exploit." Researchers are still unsure if the botnet's defenses and counter attacks are a form of automation, or manually executed by the system's operators. "If you try to attach a debugger, or query sites it's reporting into, it knows and punishes you instantaneously. [Over at] SecureWorks, a chunk of it DDoS-ed [distributed-denial-of-service attacked] a researcher off the network. Every time I hear of an investigator trying to investigate, they're automatically punished. It knows it's being investigated, and it punishes them. It fights back," Corman said.

Spameater.com as well as other sites such as 419eater.com and Artists Against 419, both of which deal with 419 spam e-mail fraud, have experienced DDoS attacks, temporarily rendering them completely inoperable. The DDoS attacks consist of making massed parallel network calls to those and other target IP addresses, overloading the servers' capacities and preventing them from responding to requests. Other anti-spam and anti-fraud groups, such as the Spamhaus Project, were also attacked. The webmaster of Artists Against 419 said that the website's server succumbed after the attack increased to over 100Mbit. As the theoretical maximum is never practically attainable, the number of

machines used may have been as much as twice that many, and similar attacks were perpetrated against over a dozen anti-fraud site hosts. Jeff Chan, a spam researcher, stated, "In terms of mitigating Storm, it's challenging at best and impossible at worst since the bad guys control many hundreds of megabits of traffic. There's some evidence that they may control hundreds of Gigabits of traffic, which is enough to force some countries off the Internet."

The Storm botnet's systems also take steps to defend itself locally, on victims' computer systems. The botnet, on some compromised systems, creates a computer process on the Windows machine that notifies the Storm systems whenever a new program or other processes begin. Previously, the Storm worms locally would tell the other programs — such as anti-virus, or anti-malware software, to simply not run. However, according to IBM security research, versions of Storm also now simply "fool" the local computer system to run the hostile program successfully, but in fact, they are not doing anything. "Programs, including not just AV exes, dlls and sys files, but also software such as the P2P applications BearShare and eDonkey, will appear to run successfully, even though they didn't actually do anything, which is far less suspicious than a process that gets terminated suddenly from the outside," said Richard Cohen of Sophos. Compromised users, and related security systems, will assume that security software is running successfully when it in fact is not.

On September 17, 2007, a Republican Party website in the United States was compromised, and used to propagate the Storm worm and botnet. In October 2007, the botnet took advantage of flaws in YouTube's captcha application on its mail systems, to send targeted spam e-mails to Xbox owners with a scam involving winning a special version of the video game *Halo 3*. Other attack methods include using appealing animated images of laughing cats to get people to click on a trojan software download, and tricking users of Yahoo!'s GeoCities service to download software that was claimed to be needed to use GeoCities itself. The GeoCities attack in particular was called a "full-fledged attack vector" by Paul Ferguson of Trend Micro, and implicated members of the Russian Business Network, a well-known spam and malware service. On Christmas Eve in 2007, the Storm botnet began sending out holiday-themed messages revolving around male interest in women, with such titles as "Find Some Christmas Tail", "The Twelve Girls of Christmas," and "Mrs. Claus Is Out Tonight!" and photos of attractive women. It was described as an attempt to draw more unprotected systems into the botnet and boost its size over the holidays, when security updates from protection vendors may take longer to be distributed. A day after the e-mails with Christmas strippers were distributed, the Storm botnet operators immediately began sending new infected e-mails that claimed to wish their recipients a "Happy New Year 2008!"

In January 2008, the botnet was detected for the first time to be involved in phishing attacks against major financial institutions, targeting both Barclays and Halifax.

Encryption and sales

Around October 15, 2007, it was uncovered that portions of the Storm botnet and its variants could be for sale. This is being done by using unique security keys in the encryption of the botnet's Internet traffic and information. The unique keys will allow each segment, or sub-section of the Storm botnet, to communicate with a section that has a matching security key. However, this may also allow people to detect, track, and block Storm botnet traffic in the future, if the security keys have unique lengths and signatures. Computer security vendor Sophos has agreed with the assessment that the partitioning of the Storm botnet indicated likely resale of its services. Graham Cluley of Sophos said, "Storm's use of encrypted traffic is an interesting feature which has raised eyebrows in our lab. Its most likely use is for the cybercriminals to lease out portions of the network for misuse. It wouldn't be a surprise if the network was used for spamming, distributed denial-of-service attacks, and other malicious activities." Security experts reported that if Storm is broken up for the malware market, in the form of a "ready-to-use botnet-making spam kit", the world could see a sharp rise in the number of Storm related infections and compromised computer systems. The encryption only seems to affect systems compromised by Storm from the second week of October 2007 onwards, meaning that any of the computer systems compromised before that time frame will remain difficult to track and block.

Within days of the discovery of this segmenting of the Storm botnet, spam e-mail from the new subsection was uncovered by major security vendors. In the evening of October 17, security vendors began seeing new spam with embedded MP3 sound files, which attempted to trick victims into investing in a penny stock, as part of an illegal pump-and-dump stock scam. It was believed that this was the first-ever spam e-mail scam that made use of audio to fool victims. Unlike nearly all other Storm-related e-mails, however, these new audio stock scam messages did not include any sort of virus or Storm malware payload; they simply were part of the stock scam.

In January 2008, the botnet was detected for the first time to be involved in phishing attacks against the customers of major financial institutions, targeting banking establishments in Europe including Barclays, Halifax and the Royal Bank of Scotland. The unique security keys used indicated to F-Secure that segments of the botnet were being leased.

Claimed decline of the botnet

On September 25, 2007, it was estimated that a Microsoft update to the Windows Malicious Software Removal Tool (MSRT) may have helped reduce the size of the botnet by up to 20%. The new patch, as claimed by Microsoft, removed Storm from approximately 274,372 infected systems out of 2.6 million scanned Windows systems. However, according to senior security staff at Microsoft, "the 180,000+ additional machines that have been cleaned by MSRT since the first day are likely to be home user machines that were not notably incorporated into the daily operation of the 'Storm' botnet," indicating that the MSRT cleaning may have been symbolic at best.

As of late October 2007, some reports indicated that the Storm botnet was losing the size of its Internet footprint, and was significantly reduced in size. Brandon Enright, a University of California at San Diego security analyst, estimated that the botnet had by late October fallen to a size of approximately 160,000 compromised systems, from Enright's previous estimated high in July 2007 of 1,500,000 systems. Enright noted, however, that the botnet's composition was constantly changing, and that it was still actively defending itself against attacks and observation. "If you're a researcher and you hit the pages hosting the malware too much… there is an automated process that automatically launches a denial of service [attack] against you," he said, and added that his research caused a Storm botnet attack that knocked part of the UC San Diego network offline.

The computer security company McAfee is reported as saying that the Storm Worm would be the basis of future attacks. Craig Schmugar, a noted security expert who discovered the Mydoom worm, called the Storm botnet a trend-setter, which has led to more usage of similar tactics by criminals. One such derivative botnet has been dubbed the "Celebrity Spam Gang", due to their use of similar technical tools as the Storm botnet controllers. Unlike the sophisticated social engineering that the Storm operators use to entice victims, however, the Celebrity spammers make use of offers of nude images of celebrities such as Angelina Jolie and Britney Spears. Cisco Systems security experts stated in a report that they believe the Storm botnet would remain a critical threat in 2008, and said they estimated that its size remained in the "millions".

As of early 2008, the Storm botnet also found business competition in its black hat economy, in the form of Nugache, another similar botnet which was first identified in 2006. Reports have indicated a price war may be underway between the operators of both botnets, for the sale of their spam E-mail delivery. Following the Christmas and New Year's holidays bridging 2007-2008, the researchers of the German Honeynet Project reported that the Storm botnet may have increased in size by up to 20% over the holidays. The *MessageLabs Intelligence* report dated March 2008 estimates that over 20% of all spam on the Internet originates from Storm.

Present state of the botnet

The Storm botnet was sending out spam for more than two years until its decline in late 2008. One factor in this — on account of making it less interesting for the creators to maintain the botnet — may have been the Stormfucker tool, which made it possible to take control over parts of the botnet.

Stormbot 2

On April 28, 2010, McAfee made an announcement that the so-called "rumors" of a Stormbot 2 were verified. Mark Schloesser, Tillmann Werner, and Felix Leder, the German researchers who did a lot of work in analyzing the original Storm, found that around two-thirds of the “new” functions are a copy and paste from the last Storm code

base. The only thing missing is the P2P infrastructure, perhaps because of the tool which used P2P to bring down the original Storm. Honeynet blog dubbed this Stormbot 2.

Chapter 13

Daemon (Computer Software)

In Unix and other computer multitasking operating systems, a **daemon** is a computer program that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes. Typically daemons have names that end with the letter "d": for example, `syslogd`, the daemon that handles the system log, or `sshd`, which handles incoming SSH connections.

In a Unix environment, the parent process of a daemon is often (but not always) the init process (PID=1). Processes usually become daemons by forking a child process and then having their parent process immediately exit, thus causing init to adopt the child process. This is a somewhat simplified view of the process as other operations are generally performed, such as dissociating the daemon process from any controlling tty. Convenience routines such as daemon(3) exist in some UNIX systems for that purpose.

Systems often start (or "launch") daemons at boot time: they often serve the function of responding to network requests, hardware activity, or other programs by performing some task. Daemons can also configure hardware (like udevd on some GNU/Linux systems), run scheduled tasks (like cron), and perform a variety of other tasks.

Terminology

The term was coined by the programmers of MIT's Project MAC. They took the name from Maxwell's demon, an imaginary being from a famous thought experiment that constantly works in the background, sorting molecules. Unix systems inherited this terminology. A derivation from the phrase "Disk and Execution Monitor" is a backronym. Daemons are also characters in Greek mythology, some of whom handled tasks that the gods could not be bothered with. BSD and some of its derivatives have adopted a daemon as its mascot, although this mascot is actually a cute variation of the demons which appear in Christian artwork.

Types of daemons

In a strictly technical sense, a Unix-like system process is a daemon when its parent process terminates and is therefore 'adopted' by the init process (process number 1) as its parent process and has no controlling terminal. However, more commonly, a daemon may be any background process, whether a child of `init` or not.

The common method for a process to become a daemon involves:

- Dissociating from the controlling tty
- Becoming a session leader
- Becoming a process group leader
- Staying in the background by forking and exiting (once or twice). This is required sometimes for the process to become a session leader. It also allows the parent process to continue its normal execution. This idiom is sometimes summarized with the phrase "fork off and die"
- Setting the root directory ("/") as the current working directory so that the process will not keep any directory in use that may be on a mounted file system (allowing it to be unmounted).
- Changing the umask to 0 to allow open(), creat(), et al. calls to provide their own permission masks and not to depend on the umask of the caller
- Closing all inherited open files at the time of execution that are left open by the parent process, including file descriptors 0, 1 and 2 (stdin, stdout, stderr). Required files will be opened later.
- Using a logfile, the console, or /dev/null as stdin, stdout, and stderr

List of service daemons for Unix and Unix-like systems

- amd - Auto Mount Daemon
- anacron - Executed delayed cron tasks at boot time
- apmd - Advanced Power Management Daemon
- arpwatch - watches for Ethernet IP address pairings that are resolved using the ARP protocol
- atd - Runs jobs queued using the at tool
- bincimapd - An imap server daemon
- biod - Cooperates with a remote nfsd to handle client Network file system requests
- bootparmd - A Internet Bootstrap Protocol server daemon
- chttpd - An http server daemon
- configd - A daemon that maintains dynamic configuration information about the computer and its environment (mainly the network)
- crond - The task scheduler daemon
- cupsd - CUPS printer daemon
- devfsd -
- dhcpd - Dynamic Host Configuration Protocol and Internet Bootstrap Protocol Server

- drakfont - A font server daemon used in Mandrake Linux
- dpid - A plugin handler for the dillo internet web browser
- egpup -
- fetchmail - daemon to retrieve mail from servers at regular intervals
- fingerd - Provides a finger protocol server
- ftpd - FTP Server Daemon
- gated - routing daemon that handles multiple routing protocols and replaces routed and egpup
- gpm - General Purpose Mouse Daemon
- httpd - Web Server Daemon
- identd - Provides the identity of a user of a particular TCP connection
- idmapd - Translates user and group identities to names, and vice versa (for NFS4)
- inetd - Internet Superserver Daemon
- initd - Initial process Daemon
- imapd - An imap server daemon
- innd - A Usenet News Server Daemon
- ipchains - A deprecated packet forwarding / firewall daemon
- isdn - ISDN network interfacing server daemon
- kapmd - Advanced Power Management Daemon
- kblockd
- kerneld - Automatically loads and unloads kernel modules
- keventd
- keytable - Loads the appropriate keyboard map from the /etc/sysconfig/keyboard configuration file
- kheader -
- klogd -
- ksoftirqd -
- kswapd - Kernel Swap daemon
- kswapd0 -
- kudzu - Detects and configures new or changed hardware during boot
- kupdated -
- launchd - PID 1 on Mac OS X systems
- linuxconf - Startup hook for the linuxconf configuration system
- lockd
- lpd - Line Printer Daemon
- mathoptd - A small single process httpd daemon with CGI support
- mcserv - Server for the midnight command networking filesystem
- memcached - In-memory distributed object caching daemon
- mpd - Music Player Daemon
- micro httpd -
- mountd - mount daemon
- mysql - Database server daemon
- named - A DNS server daemon
- netfs - Network Filesystem Mounter
- network - Activates all network interfaces at boot time
- nfsd - Network File Sharing Daemon

- nfslock - Used to start and stop nfs file locking services
- nmbd - Network Message Block Daemon
- ntpd - Network Time Protocol service daemon
- numlock - This daemon locks the numlock key during a runlevel change
- pcmcia - Provides generic pcmcia services
- portmap -
- postfix - A mail transport agent used as a replacement for sendmail
- postgresql - Database server daemon
- random - Random number generating daemon
- retchmail - A mail retrieval agent
- rlprd - Remote line printer proxy daemon
- routed - Manages routing tables
- rpcbind - Remote Procedure Call Bind Daemon
- rpciod - Remote Procedure Call Daemon
- rquotad
- rstatd - kernel statistics server daemon
- rusersd - Allows users to find one another over the network
- rwalld - Allows users to write messages on remote terminals using rwall
- rwhod - maintains the database used by the rwho and ruptime tools
- sched - Copies process regions to swap space to reclaim physical pages of memory
- sendmail - A mail transfer agent Daemon
- smbd - Samba (an SMB Server) Daemon
- smtpd - Simple Mail Transfer Protocol Daemon
- snmpd - Simple Network Management Protocol Daemon
- sound - A sound server daemon
- squid - A web page caching proxy server daemon
- sshd - Secure Shell Server Daemon
- statd -
- swapper - Copies process regions to swap space to reclaim physical pages of memory
- syncd - Keeps the file systems synchronized with system memory
- syslogd - System logging daemon
- tcpd - Service wrapper restricts access to inetd based services through hosts.allow and hosts.deny
- telnetd - Telnet Server daemon
- usb - Manages devices attached to the universal serial bus
- uptimed - Uptime logging daemon
- utelnetd - A telnet server daemon
- vhand - The "page stealing daemon" releases pages of memory for use by other processes
- vsftpd - Very Secure FTP Daemon
- walld -
- webmin - Web based administration server daemon
- xfsd - X font server daemon
- xinetd - Enhanced Internet Superserver Daemon

- xntd - Network Time Server Daemon
- ypbind - A bind server for Network Information Services

Windows equivalent

In the Microsoft DOS environment, such programs were written as Terminate and Stay Resident (TSR) software. On Microsoft Windows NT systems, programs called *services* perform the functions of daemons. They run as processes, usually do not interact with the monitor, keyboard, and mouse, and may be launched by the operating system at boot time. With Windows 2000 and later versions, one can configure and manually start and stop Windows services using the Control Panel → Administrative Tools or typing "Services.msc" in the Run command on Start menu.

However, any Windows application can perform the role of a daemon, not just a service, and some daemons for Windows have the option of running as a normal process,, where it still has no visual output.

Mac OS equivalent

On the original Mac OS, optional features and services were provided by files loaded at startup time that patched the operating system; these were known as system extensions and control panels. Later versions of classic Mac OS augmented these with fully-fledged faceless background applications: regular applications that ran in the background. To the user, these were still described as regular system extensions.

Mac OS X, being a Unix system, has daemons. There is a category of software called *services* as well, but these are different in concept from Windows' services.

Etymology

In the general sense, daemon is an older form of the word demon, from the Greek δαίμων. In the Unix System Administration Handbook, Evi Nemeth states the following about daemons:

Many people equate the word "daemon" with the word "demon", implying some kind of satanic connection between UNIX and the underworld. This is an egregious misunderstanding. "Daemon" is actually a much older form of "demon"; daemons have no particular bias towards good or evil, but rather serve to help define a person's character or personality. The ancient Greeks' concept of a "personal daemon" was similar to the modern concept of a "guardian angel" — *eudaemonia* is the state of being helped or protected by a kindly spirit. As a rule, UNIX systems seem to be infested with both daemons and demons. (p.403)

A further characterization of the mythological symbolism is that a daemon is something which is not visible yet is always present and working its will. Plato's Socrates describes his own personal daemon to be something like the modern concept of a moral conscience:

> *The favour of the gods has given me a marvelous gift, which has never left me since my childhood. It is a voice which, when it makes itself heard, deters me from what I am about to do and never urges me on.*
>
> —Character of Socrates in *Theages*, Plato

Chapter 14

Introduction to Machine Learning

Machine learning, a branch of artificial intelligence, is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from sensor data or databases. A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as examples that illustrate relations between observed variables. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data; the difficulty lies in the fact that the set of all possible behaviors given all possible inputs is too large to be covered by the set of observed examples (training data). Hence the learner must generalize from the given examples, so as to be able to produce a useful output in new cases. Machine Learning, like all subjects in artificial intelligence, require cross-disciplinary proficiency in several areas, such as probability theory, statistics, pattern recognition, cognitive science, data mining, adaptive control, computational neuroscience and theoretical computer science.

Definition

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

Generalization

The core objective of a learner is to generalize from its experience. The training examples from its experience come from some generally unknown probability distribution and the learner has to extract from them something more general, something about that distribution, that allows it to produce useful answers in new cases.

Human interaction

Some machine learning systems attempt to eliminate the need for human intuition in data analysis, while others adopt a collaborative approach between human and machine. Human intuition cannot, however, be entirely eliminated, since the system's designer must specify how the data is to be represented and what mechanisms will be used to search for a characterization of the data.

Algorithm types

Machine learning algorithms are organized into a taxonomy, based on the desired outcome of the algorithm.

- **Supervised learning** generates a function that maps inputs to desired outputs. For example, in a classification problem, the learner approximates a function mapping a vector into classes by looking at input-output examples of the function.
- **Unsupervised learning** models a set of inputs, like clustering.
- **Semi-supervised learning** combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- **Reinforcement learning** learns how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback in the form of rewards that guides the learning algorithm.
- **Transduction** tries to predict new outputs based on training inputs, training outputs, and test inputs.
- **Learning to learn** learns its own inductive bias based on previous experience.

Theory

The computational analysis of machine learning algorithms and their performance is a branch of theoretical computer science known as computational learning theory. Because training sets are finite and the future is uncertain, learning theory usually does not yield absolute guarantees of the performance of algorithms. Instead, probabilistic bounds on the performance are quite common.

In addition to performance bounds, computational learning theorists study the time complexity and feasibility of learning. In computational learning theory, a computation is considered feasible if it can be done in polynomial time. There are two kinds of time complexity results. Positive results show that a certain class of functions can be learned in polynomial time. Negative results show that certain classes cannot be learned in polynomial time.

There are many similarities between machine learning theory and statistics, although they use different terms.

Approaches

Decision tree learning

Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value.

Association rule learning

Association rule learning is a method for discovering interesting relations between variables in large databases.

Artificial neural networks

An artificial neural network (ANN), usually called "neural network" (NN), is a mathematical model or computational model that tries to simulate the structure and/or functional aspects of biological neural networks. It consists of an interconnected group of artificial neurons and processes information using a connectionist approach to computation. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs or to find patterns in data.

Genetic programming

Genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

Inductive logic programming

Inductive logic programming (ILP) is an approach to rule learning using logic programming as a uniform representation for examples, background knowledge, and hypotheses. Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an ILP system will derive a hypothesized logic program which entails all the positive and none of the negative examples.

Support vector machines

Support vector machines (SVMs) are a set of related supervised learning methods used for classification and regression. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other.

Clustering

Cluster analysis or clustering is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis.

Bayesian networks

A Bayesian network, belief network or directed acyclic graphical model is a probabilistic graphical model that represents a set of random variables and their conditional independencies 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. Efficient algorithms exist that perform inference and learning.

Reinforcement learning

Reinforcement learning is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

Applications

Applications for machine learning include machine perception, computer vision, natural language processing, syntactic pattern recognition, search engines, medical diagnosis, bioinformatics, brain-machine interfaces and cheminformatics, detecting credit card fraud, stock market analysis, classifying DNA sequences, speech and handwriting recognition, object recognition in computer vision, game playing, software engineering, adaptive websites, robot locomotion, and structural health monitoring.

Machine learning techniques helped win a major software competition: In 2006, the online movie company Netflix held the first "Netflix Prize" competition to find a program to better predict user preferences and beat its existing Netflix movie recommendation system by at least 10%. The AT&T Research Team BellKor won over several other teams with their machine learning program called Pragmatic Chaos. After winning several minor prizes, it won the 2009 grand prize competition for $1 million.

Software

RapidMiner, KNIME, Weka, ODM, Shogun toolbox and Orange are software suites containing a variety of machine learning algorithms.

Chapter 15

Supervised Learning

Supervised learning is the machine learning task of inferring a function from supervised training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which is called a classifier or a regression function. The inferred function should predict the correct output value for any valid input object. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way. (Compare with unsupervised learning.) The parallel task in human and animal psychology is often referred to as concept learning.

Overview

In order to solve a given problem of supervised learning, one has to perform the following steps:

1. Determine the type of training examples. Before doing anything else, the engineer should decide what kind of data is to be used as an example. For instance, this might be a single handwritten character, an entire handwritten word, or an entire line of handwriting.
2. Gather a training set. The training set needs to be representative of the real-world use of the function. Thus, a set of input objects is gathered and corresponding outputs are also gathered, either from human experts or from measurements.
3. Determine the input feature representation of the learned function. The accuracy of the learned function depends strongly on how the input object is represented. Typically, the input object is transformed into a feature vector, which contains a number of features that are descriptive of the object. The number of features should not be too large, because of the curse of dimensionality; but should contain enough information to accurately predict the output.
4. Determine the structure of the learned function and corresponding learning algorithm. For example, the engineer may choose to use support vector machines or decision trees.

5. Complete the design. Run the learning algorithm on the gathered training set. Some supervised learning algorithms require the user to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a validation set) of the training set, or via cross-validation.
6. Evaluate the accuracy of the learned function. After parameter adjustment and learning, the performance of the resulting function should be measured on a test set that is separate from the training set.

A wide range of supervised learning algorithms is available, each with its strengths and weaknesses. There is no single learning algorithm that works best on all supervised learning problems.

There are four major issues to consider in supervised learning:

Bias-variance tradeoff

A first issue is the tradeoff between bias and variance . Imagine that we have available several different, but equally good, training data sets. A learning algorithm is biased for a particular input x if, when trained on each of these data sets, it is systematically incorrect when predicting the correct output for x. A learning algorithm has high variance for a particular input x if it predicts different output values when trained on different training sets. The prediction error of a learned classifier is related to the sum of the bias and the variance of the learning algorithm . Generally, there is a tradeoff between bias and variance. A learning algorithm with low bias must be "flexible" so that it can fit the data well. But if the learning algorithm is too flexible, it will fit each training data set differently, and hence have high variance. A key aspect of many supervised learning methods is that they are able to adjust this tradeoff between bias and variance (either automatically or by providing a bias/variance parameter that the user can adjust).

Function complexity and amount of training data

The second issue is the amount of training data available relative to the complexity of the "true" function (classifier or regression function). If the true function is simple, then an "inflexible" learning algorithm with high bias and low variance will be able to learn it from a small amount of data. But if the true function is highly complex (e.g., because it involves complex interactions among many different input features and behaves differently in different parts of the input space), then the function will only be learnable from a very large amount of training data and using a "flexible" learning algorithm with low bias and high variance. Good learning algorithms therefore automatically adjust the bias/variance tradeoff based on the amount of data available and the apparent complexity of the function to be learned.

Dimensionality of the input space

A third issue is the dimensionality of the input space. If the input feature vectors have very high dimension, the learning problem can be difficult even if the true function only

depends on a small number of those features. This is because the many "extra" dimensions can confuse the learning algorithm and cause it to have high variance. Hence, high input dimensionality typically requires tuning the classifier to have low variance and high bias. In practice, if the engineer can manually remove irrelevant features from the input data, this is likely to improve the accuracy of the learned function. In addition, there are many algorithms for feature selection that seek to identify the relevant features and discard the irrelevant ones. This is an instance of the more general strategy of dimensionality reduction, which seeks to map the input data into a lower dimensional space prior to running the supervised learning algorithm.

Noise in the output values

A fourth issue is the degree of noise in the desired output values (the supervisory targets). If the desired output values are often incorrect (because of human error or sensor errors), then the learning algorithm should not attempt to find a function that exactly matches the training examples. This is another case where it is usually best to employ a high bias, low variance classifier.

Other factors to consider

Other factors to consider when choosing and applying a learning algorithm include the following:

1. Heterogeneity of the data. If the feature vectors include features of many different kinds (discrete, discrete ordered, counts, continuous values), some algorithms are easier to apply than others. Many algorithms, including Support Vector Machines, linear regression, logistic regression, neural networks, and nearest neighbor methods, require that the input features be numerical and scaled to similar ranges (e.g., to the [-1,1] interval). Methods that employ a distance function, such as nearest neighbor methods and support vector machines with Gaussian kernels, are particularly sensitive to this. An advantage of decision trees is that they easily handle heterogeneous data.
2. Redundancy in the data. If the input features contain redundant information (e.g., highly correlated features), some learning algorithms (e.g., linear regression, logistic regression, and distance based methods) will perform poorly because of numerical instabilities. These problems can often by solved by imposing some form of regularization.
3. Presence of interactions and non-linearities. If each of the features makes an independent contribution to the output, then algorithms based on linear functions (e.g., linear regression, logistic regression, Support Vector Machines, naive Bayes) and distance functions (e.g., nearest neighbor methods, support vector machines with Gaussian kernels) generally perform well. However, if there are complex interactions among features, then algorithms such as decision trees and neural networks work better, because they are specifically designed to discover these interactions. Linear methods can also be applied, but the engineer must manually specify the interactions when using them.

When considering a new application, the engineer can compare multiple learning algorithms and experimentally determine which one works best on the problem at hand (Tuning the performance of a learning algorithm can be very time-consuming. Given fixed resources, it is often better to spend more time collecting additional training data and more informative features than it is to spend extra time tuning the learning algorithms.

The most widely used learning algorithms are Support Vector Machines, linear regression, logistic regression, naive Bayes, linear discriminant analysis, decision trees, k-nearest neighbor algorithm, and Neural Networks (Multilayer perceptron).

How supervised learning algorithms work

Given a set of training examples of the form $\{(x_1, y_1), ..., (x_N,\ y_N)\}$, a learning algorithm seeks a function $g : X \to Y$, where X is the input space and Y is the output space. The function g is an element of some space of possible functions G, usually called the hypothesis space. It is sometimes convenient to represent g using a scoring function $f : X \times Y \to \mathbb{R}$ such that g is defined as returning the y value that gives the highest score: $g(x) = \arg\max_y\ f(x, y)$. Let F denote the space of scoring functions.

Although G and F can be any space of functions, many learning algorithms are probabilistic models where g takes the form of a conditional probability model g(x) = P(y | x), or f takes the form of a joint probability model f(x,y) = P(x,y). For example, naive Bayes and linear discriminant analysis are joint probability models, whereas logistic regression is a conditional probability model.

There are two basic approaches to choosing f or g: empirical risk minimization and structural risk minimization . Empirical risk minimization seeks the function that best fits the training data. Structural risk minimize includes a penalty function that controls the bias/variance tradeoff.

In both cases, it is assumed that the training set consists of a sample of independent and identically distributed pairs, $(x_i,\ y_i)$. In order to measure how well a function fits the training data, a loss function $L : Y \times Y \to \mathbb{R}^{\geq 0}$ is defined. For training example $(x_i,\ y_i)$, the loss of predicting the value $\hat{y}$ is $L(y_i, \hat{y})$.

The risk R(g) of function g is defined as the expected loss of g. This can be estimated from the training data as

$$R_{emp}(g) = \frac{1}{N} \sum_i L(y_i, g(x_i))$$

.

Empirical risk minimization

Empirical risk minimization (ERM) is a principle in statistical learning theory which defines a family of learning algorithms and is used to give theoretical bounds on the performance of learning algorithms.

Background

Consider the following situation, which is a general setting of many supervised learning problems. We have two spaces of objects X and Y and would like to learn a function $h : X \to Y$ (often called hypothesis) which outputs an object $y \in Y$, given $x \in X$. To do so, we have in our disposal a training set of a few examples $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$ is an input and $y_i \in Y$ is the corresponding response that we wish to get from $h(x_i)$.

To put it more formally, we assume that there is a joint probability distribution P(x,y) over X and Y, and that the training set consists of m instances $(x_1, y_1), \ldots, (x_m, y_m)$ drawn i.i.d. from P(x,y). Note that the assumption of a joint probability distribution allows us to model uncertainty in predictions (e.g. from noise in data) because y is not a deterministic function of x, but rather a random variable with conditional distribution P(y | x) for a fixed x.

We also assume that we are given a non-negative real-valued loss function $L(\hat{y}, y)$ which measures how different is the prediction $\hat{y}$ of a hypothesis from the true outcome y. The risk associated with hypothesis h(x) is then defined as the expectation of the loss function:

$$R(h) = \mathbf{E}[L(h(x), y)] = \int L(h(x), y)\, dP(x, y).$$

A loss function commonly used in theory is the 0-1 loss function: $L(\hat{y}, y) = I(\hat{y} \neq y)$, where I(...) is the indicator notation.

The ultimate goal of a learning algorithm is to find a hypothesis h^* among a fixed class of functions $\mathcal{H}$ for which the risk R(h) is minimal:

$$h^* = \arg\min_{h \in \mathcal{H}} R(h).$$

Empirical risk minimization

In general, the risk R(h) cannot be computed because the distribution P(x,y) is unknown to the learning algorithm (this situation is referred to as agnostic learning). However, we

can compute an approximation, called empirical risk, by averaging the loss function on the training set:

$$R_{\text{emp}}(h) = \frac{1}{m}\sum_{i=1}^{m} L(h(x_i), y_i).$$

Empirical risk minimization principle states that the learning algorithm should choose a hypothesis $\hat{h}$ which minimizes the empirical risk:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} R_{\text{emp}}(h).$$

Thus the learning algorithm defined by ERM principle consists in solving the above optimization problem.

Properties

Computational complexity

Empirical risk minimization for a classification problem with 0-1 loss function is known to be an NP-hard problem even for such relatively simple class of functions as linear classifiers. Though, it can be solved efficiently when minimal empirical risk is zero, i.e. data is linearly separable.

In practice, machine learning algorithms cope with that either by employing a convex approximation to 0-1 loss function (like hinge loss for SVM), which is easier to optimize, or by posing assumptions on the distribution P(x,y) (and thus stop being agnostic learning algorithms to which the above result applies,)

In empirical risk minimization, the supervised learning algorithm seeks the function g that minimizes R(g). Hence, a supervised learning algorithm can be constructed by applying an optimization algorithm to find g.

When g is a conditional probability distribution P(y | x) and the loss function is the negative log likelihood: $L(y, \hat{y}) = -\log P(y|x)$, then empirical risk minimization is equivalent to maximum likelihood estimation.

When G contains many candidate functions or the training set is not sufficiently large, empirical risk minimization leads to high variance and poor generalization. The learning algorithm is able to memorize the training examples without generalizing well. This is called overfitting.

Structural risk minimization

Structural risk minimization seeks to prevent overfitting by incorporating a regularization penalty into the optimization. The regularization penalty can be viewed as implementing a form of Occam's razor that prefers simpler functions over more complex ones.

A wide variety of penalties have been employed that correspond to different definitions of complexity. For example, consider the case where the function g is a linear function of the form

$$g(x) = \sum_{j=1}^{d} \beta_j x_j$$

.

A popular regularization penalty is $\sum_j \beta_j^2$, which is the squared Euclidean norm of the weights, also known as the L_2 norm. Other norms include the L_1 norm,

$$\sum_j |\beta_j|$$

, and the L_0 norm, which is the number of non-zero β_js. The penalty will be denoted by C(g).

The supervised learning optimization problem is to find the function g that minimizes

$$J(g) = R_{emp}(g) + \lambda C(g).$$

The parameter λ controls the bias-variance tradeoff. When λ = 0, this gives empirical risk minimization with low bias and high variance. When λ is large, the learning algorithm will have high bias and low variance. The value of λ can be chosen empirically via cross validation.

The complexity penalty has a Bayesian interpretation as the negative log prior probability of g, −logP(g), in which case J(g) is the posterior probabability of g.

Generative training

The training methods described above are discriminative training methods, because they seek to find a function g that discriminates well between the different output values. For the special case where f(x,y) = P(x,y) is a joint probability distribution and the loss function is the negative log likelihood

$$-\sum_{i} \log P(x_i, y_i),$$

a risk minimization algorithm is said to perform generative training, because f can be regarded as a generative model that explains how the data were generated. Generative training algorithms are often simpler and more computationally efficient than discriminative training algorithms. In some cases, the solution can be computed in closed form as in naive Bayes and linear discriminant analysis.

Generalizations of supervised learning

There are several ways in which the standard supervised learning problem can be generalized:

Semi-supervised learning

In computer science, **semi-supervised learning** is a class of machine learning techniques that make use of both labeled and unlabeled data for training - typically a small amount of labeled data with a large amount of unlabeled data. Semi-supervised learning falls between unsupervised learning (without any labeled training data) and supervised learning (with completely labeled training data). Many machine-learning researchers have found that unlabeled data, when used in conjunction with a small amount of labeled data, can produce considerable improvement in learning accuracy. The acquisition of labeled data for a learning problem often requires a skilled human agent to manually classify training examples. The cost associated with the labeling process thus may render a fully labeled training set infeasible, whereas acquisition of unlabeled data is relatively inexpensive. In such situations, semi-supervised learning can be of great practical value.

One example of a semi-supervised learning technique is co-training, in which two or possibly more learners are each trained on a set of examples, but with each learner using a different, and ideally independent, set of features for each example.

An alternative approach is to model the joint probability distribution of the features and the labels. For the unlabelled data the labels can then be treated as 'missing data'. Techniques that handle missing data, such as Gibbs sampling or the EM algorithm, can then be used to estimate the parameters of the model.

Active learning (machine learning)

Active learning is a form of supervised machine learning in which the learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points. In statistics literature it is sometimes also called optimal experimental design.

There are situations in which unlabeled data is abundant but labeling data is expensive. In such a scenario the learning algorithm can actively query the user/teacher for labels. This type of iterative supervised learning is called active learning. Since the learner chooses the examples, the number of examples to learn a concept can often be much lower than the number required in normal supervised learning. With this approach there is a risk that the algorithm might focus on unimportant or even invalid examples.

Active learning can be especially useful in biological research problems such as Protein engineering where a few proteins have been discovered with a certain interesting function and one wishes to determine which of many possible mutants to make next that will have a similar function .

Definitions

Let T be the total set of all data under consideration. For example, in a protein engineering problem, T would include all proteins that are known to have a certain interesting activity and all additional proteins that one might want to test for that activity.

During each iteration, i, T is broken up into three subsets

1. $\mathbf{T}_{K,i}$: Data points where the label is **known**.
2. $\mathbf{T}_{U,i}$: Data points where the label is **unknown**.
3. $\mathbf{T}_{C,i}$: A subset of $T_{U,i}$ that is **chosen** to be labeled.

Most of the current research in active learning involves the best method to chose the data points for $T_{C,i}$.

Minimum Marginal Hyperplane

Some active learning algorithms are built upon Support vector machines (SVMs) and exploit the structure of the SVM to determine which data points to label. Such methods usually calculate the margin, W, of each unlabeled datum in $T_{U,i}$ and treat W as an n-dimensional distance from that datum to separating hyperplane.

Minimum Marginal Hyperplane methods assume that the data with the smallest W are those that the SVM is most uncertain about and therefore should be placed in $T_{C,i}$ to be labeled. Other similar methods, such as Maximum Marginal Hyperplane, choose data with the largest W. Tradeoff methods choose a mix of the smallest and largest Ws.

Maximum Curiosity

Another active learning method, that typically learns a data set with fewer examples than Minimum Marginal Hyperplane but is more computationally intensive and only works for discrete classifiers is Maximum Curiosity.

Maximum curiosity takes each unlabeled datum in $T_{U,i}$ and assumes all possible labels that datum might have. This datum with each assumed class is added to $T_{K,i}$ and then the new $T_{K,i}$ is cross-validated. It is assumed that when the datum is paired up with its correct label, the cross-validated accuracy (or correlation coefficient) of $T_{K,i}$ will most improve. The datum with the most improved accuracy is placed in $T_{C,i}$ to be labeled

Structured prediction

Structured prediction is an umbrella term for machine learning and regression techniques that involve predicting structured objects. For example, the problem of translating a natural language sentence into a semantic representation such as a parse tree can be seen as a structured prediction problem in which the structured output domain is the set of all possible parse trees. Structured prediction generalizes supervised learning where the output domain is usually a small or simple set.

Probabilistic graphical models form a large class of structured prediction models. In particular, Bayesian networks and random fields are popularly used to solve structured prediction problems in a wide variety of application domains including bioinformatics, natural language processing, speech recognition, and computer vision.

Similar to commonly used supervised learning techniques, structured prediction models are typically trained by means of observed data in which the true prediction value is used to adjust model parameters. Due to the complexity of the model and the interrelations of predicted variables the process of prediction using a trained model and of training itself is often computationally infeasible and approximate inference and learning methods are used.

Another commonly used term for structured prediction is structured output learning.

Chapter 16

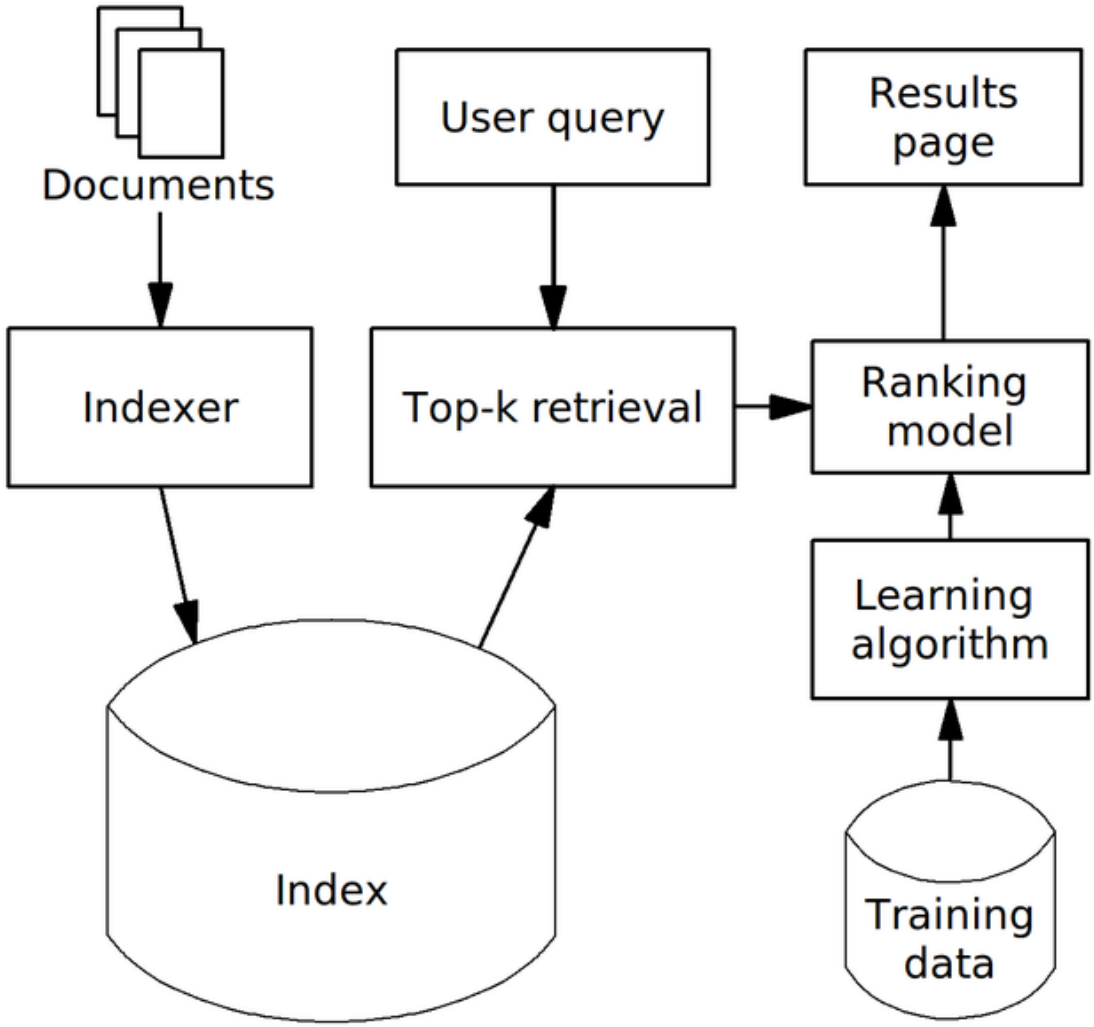
Learning to Rank

Learning to rank or **machine-learned ranking** (MLR) is a type of supervised or semi-supervised machine learning problem in which the goal is to automatically construct a ranking model from training data. Training data consists of lists of items with some partial order specified between items in each list. This order is typically induced by giving a numerical or ordinal score or a binary judgment (e.g. "relevant" or "not relevant") for each item. Ranking model's purpose is to rank, i.e. produce a permutation of items in new, unseen lists in a way, which is "similar" to rankings in the training data in some sense.

Learning to rank is a relatively new research area which has emerged in the past decade.

Applications

In information retrieval



A possible architecture of a machine-learned search engine.

Ranking is a central part of many information retrieval problems, such as document retrieval, collaborative filtering, sentiment analysis, computational advertising (online ad placement).

When applied to document retrieval, the task of learning to rank is to construct a ranking function for a search engine. In this case each list in training data represents documents which match a search query and they are ordered according to relevance to the query.

A possible architecture of a machine-learned search engine is shown in the figure to the right.

Training data consists of queries and documents matching them together with relevance degree of each match. It may be prepared manually by human assessors (or raters, as Google calls them), who check results for some queries and determine relevance of each result. It is not feasible to check relevance of all documents, and so typically a technique called pooling is used — only top few documents, retrieved by some existing ranking models are checked. Alternatively, training data may be derived automatically by analyzing clickthrough logs (i.e. search results which got clicks from users), query chains, or such search engines' features as Google's.

Training data is used by a learning algorithm to produce a ranking model which computes relevance of documents for actual queries.

Typically, users expect a search query to complete in a short time (such as a few hundred milliseconds for web search), which makes it impossible to evaluate a complex ranking model on each document in the corpus, and so a two-phase scheme is used. First, a small number of potentially relevant documents are identified using simpler retrieval models which permit fast query evaluation, such as vector space model, boolean model, weighted AND , BM25. This phase is called top-k document retrieval and many good heuristics were proposed in the literature to accelerate it, such as using document's static quality score and tiered indexes. In the second phase, a more accurate but computationally expensive machine-learned model is used to re-rank these documents.

In other areas

Learning to rank algorithms have been applied in areas other than information retrieval:

- In machine translation for ranking a set of hypothesized translations;
- In computational biology for ranking candidate 3-D structures in protein structure prediction problem.
- In proteomics for the identification of frequent top scoring peptides.

Feature vectors

For convenience of MLR algorithms, query-document pairs are usually represented by numerical vectors, which are called feature vectors. Such approach is sometimes called bag of features and is analogous to bag of words and vector space model used in information retrieval for representation of documents.

Components of such vectors are called features, factors or ranking signals. They may be divided into three groups (features from document retrieval are shown as examples):

- Query-independent or static features — those features, which depend only on the document, but not on the query. For example, PageRank or document's length. Such features can be precomputed in off-line mode during indexing. They may be used to compute document's static quality score (or static rank), which is often used to speed up search query evaluation.

- Query-dependent or dynamic features — those features, which depend both on the contents of the document and the query, such as TF-IDF score or other non-machine-learned ranking functions.
- Query features, which depend only on the query. For example, the number of words in a query.

Some examples of features, which were used in the well-known LETOR dataset:

- TF, TF-IDF, BM25, and language modeling scores of document's zones (title, body, anchors text, URL) for a given query;
- Lengths and IDF sums of document's zones;
- Document's PageRank, HITS ranks and their variants.

Selecting and designing good features is an important area in machine learning, which is called feature engineering.

Evaluation measures

There are several measures (metrics) which are commonly used to judge how well an algorithm is doing on training data and to compare performance of different MLR algorithms. Often a learning-to-rank problem is reformulated as an optimization problem with respect to one of these metrics.

Examples of ranking quality measures:

Information retrieval

Information retrieval (**IR**) is the science of searching for documents, for information within documents, and for metadata about documents, as well as that of searching relational databases and the World Wide Web. There is overlap in the usage of the terms data retrieval, document retrieval, information retrieval, and text retrieval, but each also has its own body of literature, theory, praxis, and technologies. IR is interdisciplinary, based on computer science, mathematics, library science, information science, information architecture, cognitive psychology, linguistics, and statistics.

Automated information retrieval systems are used to reduce what has been called "information overload". Many universities and public libraries use IR systems to provide access to books, journals and other documents. Web search engines are the most visible IR applications.

History

> “But do you know that, although I have kept the diary [on a phonograph] for months past, it never once struck me how I was going to find any particular part of it in case I wanted to look it up?”
>
> —Dr Seward, Bram Stoker's Dracula, 1897

The idea of using computers to search for relevant pieces of information was popularized in the article As We May Think by Vannevar Bush in 1945. The first automated information retrieval systems were introduced in the 1950s and 1960s. By 1970 several different techniques had been shown to perform well on small text corpora such as the Cranfield collection (several thousand documents). Large-scale retrieval systems, such as the Lockheed Dialog system, came into use early in the 1970s.

In 1992, the US Department of Defense along with the National Institute of Standards and Technology (NIST), cosponsored the Text Retrieval Conference (TREC) as part of the TIPSTER text program. The aim of this was to look into the information retrieval community by supplying the infrastructure that was needed for evaluation of text retrieval methodologies on a very large text collection. This catalyzed research on methods that scale to huge corpora. The introduction of web search engines has boosted the need for very large scale retrieval systems even further.

The use of digital methods for storing and retrieving information has led to the phenomenon of digital obsolescence, where a digital resource ceases to be readable because the physical media, the reader required to read the media, the hardware, or the software that runs on it, is no longer available. The information is initially easier to retrieve than if it were on paper, but is then effectively lost.

Timeline

- Before the **1900s**

  1880s: Herman Hollerith invents the recording of data on a machine readable medium.

1890 Hollerith cards, key punches and tabulators used to process the 1890 US Census data.

- **1940s–1950s**

  late 1940s: The US military confronted problems of indexing and retrieval of wartime scientific research documents captured from Germans.
  1945: Vannevar Bush's As We May Think appeared in Atlantic Monthly.
  1947: Hans Peter Luhn (research engineer at IBM since 1941) began work on a mechanized punch card-based system for searching chemical compounds.
  1950s: Growing concern in the US for a "science gap" with the USSR motivated, encouraged funding and provided a backdrop for mechanized literature searching systems (Allen Kent et al.) and the invention of citation indexing (Eugene Garfield).
  1950: The term "information retrieval" appears to have been coined by Calvin Mooers.
  1951: Philip Bagley conducted the earliest experiment in computerized document retrieval in a master thesis at MIT.
  1955: Allen Kent joined Case Western Reserve University, and eventually became associate director of the Center for Documentation and Communications Research. That same year, Kent and colleagues published a paper in American Documentation describing the precision and recall measures as well as detailing a proposed "framework" for evaluating an IR system which included statistical sampling methods for determining the number of relevant documents not retrieved.
  1958: International Conference on Scientific Information Washington DC included consideration of IR systems as a solution to problems identified. See: Proceedings of the International Conference on Scientific Information, 1958 (National Academy of Sciences, Washington, DC, 1959)
  1959: Hans Peter Luhn published "Auto-encoding of documents for information retrieval."

- **1960s**:

  early 1960s: Gerard Salton began work on IR at Harvard, later moved to Cornell.
  1960: Melvin Earl (Bill) Maron and John Lary Kuhns published "On relevance, probabilistic indexing, and information retrieval" in the Journal of the ACM 7(3):216–244, July 1960.
  1962:

  - Cyril W. Cleverdon published early findings of the Cranfield studies, developing a model for IR system evaluation. See: Cyril W. Cleverdon, "Report on the Testing and Analysis of an Investigation into the Comparative Efficiency of Indexing Systems". Cranfield Collection of Aeronautics, Cranfield, England, 1962.
  - Kent published Information Analysis and Retrieval.

1963:

- Weinberg report "Science, Government and Information" gave a full articulation of the idea of a "crisis of scientific information." The report was named after Dr. Alvin Weinberg.
- Joseph Becker and Robert M. Hayes published text on information retrieval. Becker, Joseph; Hayes, Robert Mayo. Information storage and retrieval: tools, elements, theories. New York, Wiley (1963).

1964:

- Karen Spärck Jones finished her thesis at Cambridge, Synonymy and Semantic Classification, and continued work on computational linguistics as it applies to IR.
- The National Bureau of Standards sponsored a symposium titled "Statistical Association Methods for Mechanized Documentation." Several highly significant papers, including G. Salton's first published reference (we believe) to the SMART system.

mid-1960s:

- National Library of Medicine developed MEDLARS Medical Literature Analysis and Retrieval System, the first major machine-readable database and batch-retrieval system.
- Project Intrex at MIT.

1965: J. C. R. Licklider published Libraries of the Future.

1966: Don Swanson was involved in studies at University of Chicago on Requirements for Future Catalogs.

late 1960s: F. Wilfrid Lancaster completed evaluation studies of the MEDLARS system and published the first edition of his text on information retrieval.

1968:

- Gerard Salton published Automatic Information Organization and Retrieval.
- John W. Sammon, Jr.'s RADC Tech report "Some Mathematics of Information Storage and Retrieval..." outlined the vector model.

1969: Sammon's "A nonlinear mapping for data structure analysis" (IEEE Transactions on Computers) was the first proposal for visualization interface to an IR system.

- **1970s**

early 1970s:

- First online systems—NLM's AIM-TWX, MEDLINE; Lockheed's Dialog; SDC's ORBIT.
- Theodor Nelson promoting concept of hypertext, published Computer Lib/Dream Machines.

1971: Nicholas Jardine and Cornelis J. van Rijsbergen published "The use of hierarchic clustering in information retrieval", which articulated the "cluster hypothesis." (Information Storage and Retrieval, 7(5), pp. 217–240, December 1971)

1975: Three highly influential publications by Salton fully articulated his vector processing framework and term discrimination model:

- A Theory of Indexing (Society for Industrial and Applied Mathematics)
- A Theory of Term Importance in Automatic Text Analysis (JASIS v. 26)
- A Vector Space Model for Automatic Indexing (CACM 18:11)

1978: The First ACM SIGIR conference.
1979: C. J. van Rijsbergen published Information Retrieval (Butterworths). Heavy emphasis on probabilistic models.

- **1980s**

1980: First international ACM SIGIR conference, joint with British Computer Society IR group in Cambridge.

1982: Nicholas J. Belkin, Robert N. Oddy, and Helen M. Brooks proposed the ASK (Anomalous State of Knowledge) viewpoint for information retrieval. This was an important concept, though their automated analysis tool proved ultimately disappointing.

1983: Salton (and Michael J. McGill) published Introduction to Modern Information Retrieval (McGraw-Hill), with heavy emphasis on vector models.

mid-1980s: Efforts to develop end-user versions of commercial IR systems.

1985–1993: Key papers on and experimental systems for visualization interfaces. Work by Donald B. Crouch, Robert R. Korfhage, Matthew Chalmers, Anselm Spoerri and others.

1989: First World Wide Web proposals by Tim Berners-Lee at CERN.

- **1990s**

1992: First TREC conference.

1997: Publication of Korfhage's Information Storage and Retrieval with emphasis on visualization and multi-reference point systems.

late 1990s: Web search engines implementation of many features formerly found only in experimental IR systems. Search engines become the most common and maybe best instantiation of IR models, research, and implementation.

Overview

An information retrieval process begins when a user enters a query into the system. Queries are formal statements of information needs, for example search strings in web search engines. In information retrieval a query does not uniquely identify a single object in the collection. Instead, several objects may match the query, perhaps with different degrees of relevancy.

An object is an entity that is represented by information in a database. User queries are matched against the database information. Depending on the application the data objects may be, for example, text documents, images , audio, mind maps or videos. Often the documents themselves are not kept or stored directly in the IR system, but are instead represented in the system by document surrogates or metadata.

Most IR systems compute a numeric score on how well each object in the database match the query, and rank the objects according to this value. The top ranking objects are then shown to the user. The process may then be iterated if the user wishes to refine the query.

Performance measures

Many different measures for evaluating the performance of information retrieval systems have been proposed. The measures require a collection of documents and a query. All common measures described here assume a ground truth notion of relevancy: every document is known to be either relevant or non-relevant to a particular query. In practice queries may be ill-posed and there may be different shades of relevancy.

Precision

Precision is the fraction of the documents retrieved that are relevant to the user's information need.

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

In binary classification, precision is analogous to positive predictive value. Precision takes all retrieved documents into account. It can also be evaluated at a given cut-off

rank, considering only the topmost results returned by the system. This measure is called precision at n or P@n.

Note that the meaning and usage of "precision" in the field of Information Retrieval differs from the definition of accuracy and precision within other branches of science and technology.

Recall

Recall is the fraction of the documents that are relevant to the query that are successfully retrieved.

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

In binary classification, recall is called sensitivity. So it can be looked at as the probability that a relevant document is retrieved by the query.

It is trivial to achieve recall of 100% by returning all documents in response to any query. Therefore recall alone is not enough but one needs to measure the number of non-relevant documents also, for example by computing the precision.

Fall-Out

The proportion of non-relevant documents that are retrieved, out of all non-relevant documents available:

$$\text{fall-out} = \frac{|\{\text{non-relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{non-relevant documents}\}|}$$

In binary classification, fall-out is closely related to specificity (1 − specificity). It can be looked at as the probability that a non-relevant document is retrieved by the query.

It is trivial to achieve fall-out of 0% by returning zero documents in response to any query.

F-measure

The weighted harmonic mean of precision and recall, the traditional F-measure or balanced F-score is:

$$F = \frac{2 \cdot \text{precision} \cdot \text{recall}}{(\text{precision} + \text{recall})}.$$

This is also known as the F_1 measure, because recall and precision are evenly weighted.

The general formula for non-negative real β is:

$$F_\beta = \frac{(1 + \beta^2) \cdot (\text{precision} \cdot \text{recall})}{(\beta^2 \cdot \text{precision} + \text{recall})}$$

Two other commonly used F measures are the F_2 measure, which weights recall twice as much as precision, and the $F_{0.5}$ measure, which weights precision twice as much as recall.

The F-measure was derived by van Rijsbergen (1979) so that F_β "measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision". It is based on van Rijsbergen's effectiveness measure $E = 1 - (1 / (\alpha / P + (1 - \alpha) / R))$. Their relationship is $F_\beta = 1 - E$ where $\alpha = 1 / (\beta^2 + 1)$.

Average precision

Precision and recall are single-value metrics based on the whole list of documents returned by the system. For systems that return a ranked sequence of documents, it is desirable to also consider the order in which the returned documents are presented. Average precision emphasizes ranking relevant documents higher. It is the average of precisions computed at the point of each of the relevant documents in the ranked sequence:

$$\text{AveP} = \frac{\sum_{r=1}^{N} (P(r) \times \text{rel}(r))}{\text{number of relevant documents}}$$

where r is the rank, N the number retrieved, rel() a binary function on the relevance of a given rank, and P(r) precision at a given cut-off rank:

$$\text{P(r)} = \frac{|\{\text{relevant retrieved documents of rank r or less}\}|}{r}$$

This metric is also sometimes referred to geometrically as the area under the Precision-Recall curve.

Note that the denominator (number of relevant documents) is the number of relevant documents in the entire collection, so that the metric reflects performance over all relevant documents, regardless of a retrieval cutoff.

Mean average precision

Mean average precision for a set of queries is the mean of the average precision scores for each query.

$$\mathrm{MAP} = \frac{\sum_{q=1}^{Q} \mathrm{AveP(q)}}{Q}$$

where Q is the number of queries.

Discounted cumulative gain

DCG uses a graded relevance scale of documents from the result set to evaluate the usefulness, or gain, of a document based on its position in the result list. The premise of DCG is that highly relevant documents appearing lower in a search result list should be penalized as the graded relevance value is reduced logarithmically proportional to the position of the result.

The DCG accumulated at a particular rank position p is defined as:

$$\mathrm{DCG_p} = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i}$$

Since result set may vary in size among different queries or systems, to compare performances the normalised version of DCG uses an ideal DCG. To this end, it sorts documents of a result list by relevance, producing an ideal DCG at position p ($IDCG_p$), which normalizes the score:

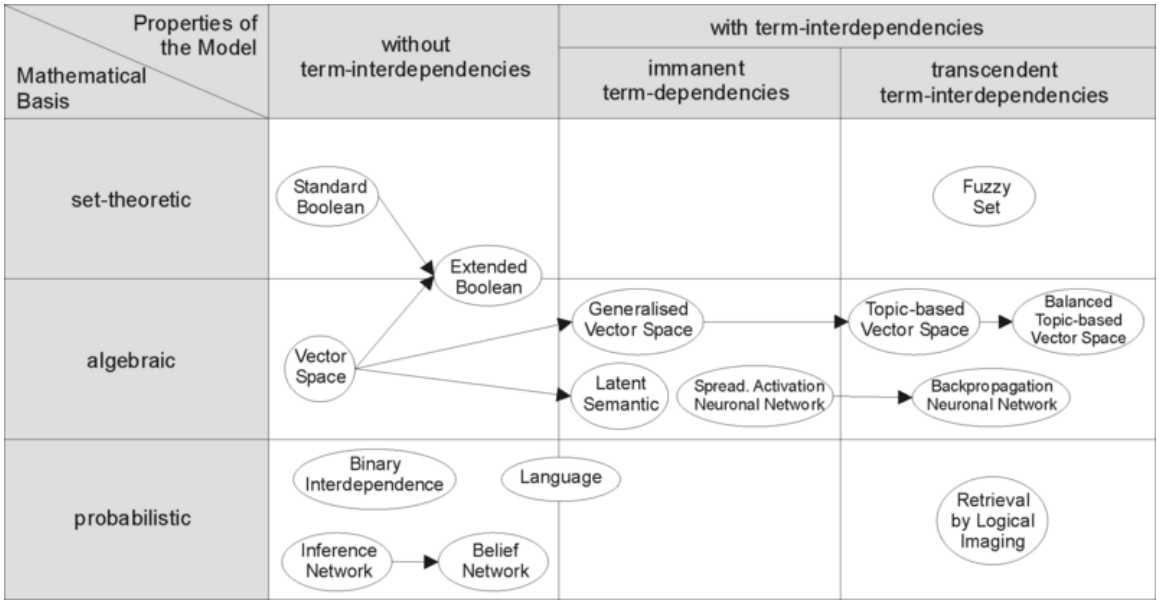
$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCGp}$$

The nDCG values for all queries can be averaged to obtain a measure of the average performance of a ranking algorithm. Note that in a perfect ranking algorithm, the DCG_p will be the same as the $IDCG_p$ producing an nDCG of 1.0. All nDCG calculations are then relative values on the interval 0.0 to 1.0 and so are cross-query comparable.

Other Measures

- Mean reciprocal rank
- Spearman's rank correlation coefficient

Model types



Categorization of IR-models (translated from German entry, original source Dominik Kuropka).

For the information retrieval to be efficient, the documents are typically transformed into a suitable representation. There are several representations. The picture on the right illustrates the relationship of some common models. In the picture, the models are categorized according to two dimensions: the mathematical basis and the properties of the model.

First dimension: mathematical basis

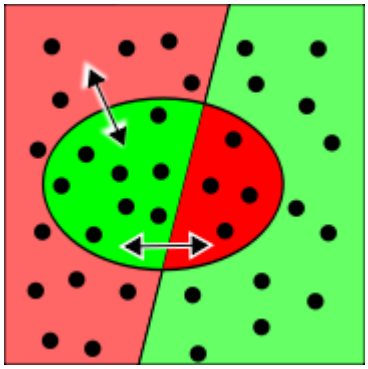
- Set-theoretic models represent documents as sets of words or phrases. Similarities are usually derived from set-theoretic operations on those sets. Common models are:
  - Standard Boolean model
  - Extended Boolean model
  - Fuzzy retrieval
- Algebraic models represent documents and queries usually as vectors, matrices, or tuples. The similarity of the query vector and document vector is represented as a scalar value.
  - Vector space model
  - Generalized vector space model
  - (Enhanced) Topic-based Vector Space Model
  - Extended Boolean model
  - Latent semantic indexing aka latent semantic analysis
- Probabilistic models treat the process of document retrieval as a probabilistic inference. Similarities are computed as probabilities that a document is relevant

for a given query. Probabilistic theorems like the Bayes' theorem are often used in these models.
    - Binary Independence Model
    - Probabilistic relevance model on which is based the okapi (BM25) relevance function
    - Uncertain inference
    - Language models
    - Divergence-from-randomness model
    - Latent Dirichlet allocation
- Machine-learned ranking models view documents as vectors of ranking features (some of which often incorporate other ranking models mentioned above) and try to find the best way to combine these features into a single relevance score by machine learning methods.

Second dimension: properties of the model

- Models without term-interdependencies treat different terms/words as independent. This fact is usually represented in vector space models by the orthogonality assumption of term vectors or in probabilistic models by an independency assumption for term variables.
- Models with immanent term interdependencies allow a representation of interdependencies between terms. However the degree of the interdependency between two terms is defined by the model itself. It is usually directly or indirectly derived (e.g. by dimensional reduction) from the co-occurrence of those terms in the whole set of documents.
- Models with transcendent term interdependencies allow a representation of interdependencies between terms, but they do not allege how the interdependency between two terms is defined. They relay an external source for the degree of interdependency between two terms. (For example a human or sophisticated algorithms.)

Precision and recall



Recall and precision depend on the outcome (oval) of a query and its relation to all relevant documents (left) and the non-relevant documents (right). The more correct results (green), the better. **Precision**: horizontal arrow. **Recall**: diagonal arrow.

Precision and **recall** are two widely used metrics for evaluating the correctness of a pattern recognition algorithm. They can be seen as extended versions of accuracy, a simple metric that computes the fraction of instances for which the correct result is returned.

When using precision and recall, the set of possible labels for a given instance is divided into two subsets, one of which is considered "relevant" for the purposes of the metric. Recall is then computed as the fraction of correct instances among all instances that actually belong to the relevant subset, while precision is the fraction of correct instances among those that the algorithm believes to belong to the relevant subset.

Precision can be seen as a measure of exactness or fidelity, whereas recall is a measure of completeness.

Introduction

As an example, in an information retrieval scenario, the instances are documents and the task is to return a set of relevant documents given a search term; or equivalently, to assign each document to one of two categories, "relevant" and "not relevant". In this case, the "relevant" documents are simply those that belong to the "relevant" category. Recall is defined as the number of relevant documents retrieved by a search divided by the total number of existing relevant documents, while precision is defined as the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search.

In a classification task, the precision for a class is the number of **true positives** (i.e. the number of items correctly labeled as belonging to the positive class) divided by the total number of elements labeled as belonging to the positive class (i.e. the sum of true positives and **false positives**, which are items incorrectly labeled as belonging to the class). Recall in this context is defined as the number of true positives divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and **false negatives**, which are items which were not labeled as belonging to the positive class but should have been).

In information retrieval, a perfect precision score of 1.0 means that every result retrieved by a search was relevant (but says nothing about whether all relevant documents were retrieved) whereas a perfect recall score of 1.0 means that all relevant documents were retrieved by the search (but says nothing about how many irrelevant documents were also retrieved).

In a classification task, a precision score of 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C (but says nothing about the

number of items from class C that were not labeled correctly) whereas a recall of 1.0 means that every item from class C was labeled as belonging to class C (but says nothing about how many other items were incorrectly also labeled as belonging to class C).

Often, there is an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other. For example, an information retrieval system (such as a search engine) can often increase its recall by retrieving more documents, at the cost of increasing number of irrelevant documents retrieved (decreasing precision). Similarly, a classification system for deciding whether or not, say, a fruit is an orange, can achieve high precision by only classifying fruits with the exact right shape and color as oranges, but at the cost of low recall due to the number of false negatives from oranges that did not quite match the specification.

Usually, precision and recall scores are not discussed in isolation. Instead, either values for one measure are compared for a fixed level at the other measure (e.g. precision at a recall level of 0.75) or both are combined into a single measure, such as the **F-measure**, which is the weighted harmonic mean of precision and recall (see below), or the Matthews correlation coefficient.

Definition (information retrieval context)

In information retrieval contexts, precision and recall are defined in terms of a set of **retrieved documents** (e.g. the list of documents produced by a web search engine for a query) and a set of **relevant documents** (e.g. the list of all documents on the internet that are relevant for a certain topic).

Precision

In the field of information retrieval, **precision** is the fraction of retrieved documents that are relevant to the search:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Precision takes all retrieved documents into account, but it can also be evaluated at a given cut-off rank, considering only the topmost results returned by the system. This measure is called **precision at n** or **P@n**.

For example for a text search on a set of documents precision is the number of correct results divided by the number of all returned results.

Precision is also used with recall, the percent of all relevant documents that is returned by the search. The two measures are sometimes used together in the F1 Score (or f-measure) to provide a single measurement for a system.

Note that the meaning and usage of "precision" in the field of Information Retrieval differs from the definition of accuracy and precision within other branches of science and technology.

Recall

Recall in Information Retrieval is the fraction of the documents that are relevant to the query that are successfully retrieved.

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

For example for text search on a set of documents recall is the number of correct results divided by the number of results that should have been returned

In binary classification, recall is called sensitivity. So it can be looked at as the probability that a relevant document is retrieved by the query.

It is trivial to achieve recall of 100% by returning all documents in response to any query. Therefore recall alone is not enough but one needs to measure the number of non-relevant documents also, for example by computing the precision.

Definition (classification context)

In the context of classification tasks, the terms **true positives**, **true negatives**, **false positives** and **false negatives** are used to compare the given classification of an item (the class label assigned to the item by a classifier) with the desired correct classification (the class the item actually belongs to). This is illustrated by the table below:

		correct result / classification	
		E1	**E2**
obtained result / classification	**E1**	**tp** (true positive)	**fp** (false positive)
	E2	**fn** (false negative)	**tn** (true negative)

Precision and recall are then defined as:

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

Recall in this context is also referred to as the True Positive Rate, other related measures used in classification include True Negative Rate and Accuracy: . True Negative Rate is also called Specificity.

$$\text{True Negative Rate} = \frac{tn}{tn + fp}$$

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

Probabilistic interpretation

It is possible to interpret precision and recall not as ratios but as probabilities:

- **Precision** is the probability that a (randomly selected) retrieved document is relevant.
- **Recall** is the probability that a (randomly selected) relevant document is retrieved in a search.

F-measure

A measure that combines precision and recall is the harmonic mean of precision and recall, the traditional F-measure or balanced F-score:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

This is also known as the F_1 measure, because recall and precision are evenly weighted.

It is a special case of the general F_β measure (for non-negative real values of β):

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}}$$

Two other commonly used F measures are the F_2 measure, which weights recall twice as much as precision, and the $F_{0.5}$ measure, which weights precision twice as much as recall.

The F-measure was derived by van Rijsbergen (1979) so that F_β "measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision". It is based on van Rijsbergen's effectiveness measure $E = 1 - (1/(\alpha/P + (1 - \alpha)/R))$. Their relationship is $F_\beta = 1 - E$ where $\alpha = 1/(\beta^2 + 1)$.

Chapter 17

Types of Machine Learning

Reinforcement learning

Inspired by old behaviorist psychology, **reinforcement learning** is an area of machine learning in computer science, concerned with how an agent ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as control theory, operations research, information theory, simulation-based optimization, statistics, and Genetic Algorithms. In the operations research and control literature the field where reinforcement learning methods are studied is called approximate dynamic programming. The problem has been studied in the theory of optimal control, though most studies there are concerned with existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov decision process (MDP), and many reinforcement learning algorithms for this context are highly related to dynamic programming techniques. The main difference to these classical techniques is that reinforcement learning algorithms do not need the knowledge of the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). The exploration vs. exploitation trade-off in reinforcement learning has been most thoroughly studied through the multi-armed bandit problem and in finite MDPs.

The basic reinforcement learning model consists of:

1. a set of environment states S;
2. a set of actions A;

3. rules of transitioning between states;
4. rules that determine the scalar immediate reward of a transition; and
5. rules that describe what the agent observes.

The rules are often stochastic. The observation typically involves the scalar immediate reward associated to the last transition. In many works, the agent is also assumed to observe the current environmental state, in which case we talk about full observability, whereas in the opposing case we talk about partial observability. Sometimes the set of actions available to the agent is restricted (e.g., you cannot spend more money than what you possess).

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation o_t, which typically includes the reward r_t. It then chooses an action a_t from the set of actions available, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t,a_t,s_{t+1}) is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can choose any action as a function of the history and it can even randomize its action selection.

When the agent's performance is compared to that of an agent which acts optimally from the beginning, the difference in performance gives rise to the notion of regret. Note that in order to act near optimally, the agent must reason about the long term consequences of its actions: In order to maximize my future income I better go to school now, although the immediate monetary reward associated with this might be negative.

Thus, reinforcement learning is particularly well suited to problems which include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, backgammon and chess.

Two components make reinforcement learning powerful: The use of samples to optimize performance and the use of function approximation to deal with large environments. Thus, reinforcement learning is most successful when the environment is big or cannot be precisely described. However, reinforcement learning methods can also be applied when the environment is big, but can be reasonably simulated, a problem studied in simulation-based optimization.

Exploration

The reinforcement learning problem as described requires clever exploration mechanisms. Randomly selecting actions is known to give rise to very poor performance. The case of (small) finite MDPs is relatively well understood by now. However, due to the lack of algorithms that would provably scale well with the number of states (or scale to problems with infinite state spaces), in practice people resort to simple exploration methods. One such method is ε-greedy, when the agent chooses the action that it believes has the best long-term effect with probability $1 - \varepsilon$, and it chooses an action uniformly at

random, otherwise. Here, $0 < \varepsilon < 1$ is a tuning parameter, which is sometimes changed, either according to a fixed schedule (making the agent explore less as time goes by), or adaptively based on some heuristics (Tokic, 2010).

Algorithms for control learning

Even if the issue of exploration is disregarded and even if the state was observable (which we assume from now on), the problem remains to find out which actions are good based on past experience.

Criterion of optimality

For simplicity, assume for a moment that the problem studied is episodic, an episode ending when some terminal state is reached. Assume further that no matter what course of actions the agent takes, termination is inevitable with probability one. Under some additional mild regularity conditions the expectation of the total reward is then well-defined, for any policy π and any initial distribution over the states. Given a fixed initial distribution μ, we can thus assign the expected return ρ^π to policy π:

$$\rho^\pi = E[R \mid \pi],$$

where the random variable R denotes the return and is defined by

$$R = \sum_{t=0}^{N-1} r_{t+1},$$

where r_{t+1} is the reward received after the t-th transition, the initial state is sampled at random from μ and actions are selected by policy π. Here, N denotes the (random) time when a terminal state is reached, i.e., the time when the episode terminates.

In the case of non-episodic problems the return is often discounted,

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1},$$

giving rise to the total expected discounted reward criterion. Here $0 \leq \gamma \leq 1$is the so-called discount-factor. Since the undiscounted return is a special case of the discounted return, from now on we will assume discounting. Although this looks innocent enough, discounting is in fact problematic if one cares about online performance. This is because discounting makes the initial time steps more important. Since a learning agent is likely to make mistakes during the first few steps after its "life" starts, no uninformed learning algorithm can achieve near-optimal performance under discounting even if the class of environments is restricted to that of finite MDPs. (This does not mean though that, given enough time, a learning agent cannot figure how to act near-optimally, if time was restarted.)

The problem then is to specify an algorithm that can be used to find a policy with maximum expected return. From the theory of MDPs it is known that, without the loss of generality, the search can be restricted to the set of the so-called stationary policies. A policy is called stationary if the action-distribution returned by it depends only the last state visited (which is part of the observation history of the agent, by our simplifying assumption). In fact, the search can be further restricted to deterministic stationary policies. A deterministic stationary policy is one which deterministically selects actions based on the current state. Since any such policy can be identified with a mapping from the set of states to the set of action, these policies can be identified with such mappings with no loss of generality.

Brute force

The naive brute force approach entails the following two steps:

1. For each possible policy, sample returns while following it
2. Choose the policy with the largest expected return

One problem with this is that the number of policies can be extremely large, or even infinite. Another is that variance of the returns might be large, in which case a large number of samples will be required to accurately estimate the return of each policy.

These problems can be ameliorated if we assume some structure and perhaps allow samples generated from one policy to influence the estimates made for another. The two main approaches for achieving this are value function estimation and direct policy search.

Value function approaches

Value function approaches attempt to find a policy that maximizes the return by maintaining a set of estimates of expected returns for some policy (usually either the "current" or the optimal one).

These methods rely on the theory of MDPs, where optimality is defined in a sense which is stronger than the above one: A policy is called optimal if it achieves the best expected return from any initial state (i.e., initial distributions play no role in this definition). Again, one can always find an optimal policy amongst stationary policies.

To define optimality in a formal manner, define the value of a policy π by

$$V^{\pi}(s) = E[R \mid s,\pi],$$

where R stands for the random return associated with following π from the initial state s. Define $V^{*}(s)$ as the maximum possible value of $V^{\pi}(s)$, where π is allowed to change:

$$V^*(s) = \sup_{\pi} V^{\pi}(s).$$

A policy which achieves these optimal values in each state is called optimal. Clearly, a policy optimal in this strong sense is also optimal in the sense that it maximizes the expected return ρ^{π}, since $\rho^{\pi} = E[V^{\pi}(S)]$, where S is a state randomly sampled from the distribution μ.

Although state-values suffice to define optimality, it will prove to be useful to define action-values. Given a state s, an action a and a policy π, the action-value of the pair (s,a) under π is defined by

$$Q^{\pi}(s,a) = E[R|s,a,\pi],$$

where, now, R stands for the random return associated with first taking action a in state s and following π, thereafter.

It is well-known from the theory of MDPs that if someone gives us Q for an optimal policy, we can always choose optimal actions (and thus act optimally) by simply choosing the action with the highest value at each state. The action-value function of such an optimal policy is called the optimal action-value function and is denoted by Q^{*}. In summary, the knowledge of the optimal action-value function alone suffices to know how to act optimally.

Assuming full knowledge of the MDP, there are two basic approaches to compute the optimal action-value function, value iteration and policy iteration. Both algorithms compute a sequence of functions Q_k ($k = 0, 1, 2, \ldots$) which converge to Q^{*}. Computing these functions involves computing expectations over the whole state-space, which is impractical for all, but the smallest (finite) MDPs, never mind the case when the MDP is unknown. In reinforcement learning methods the expectations are approximated by averaging over samples and one uses function approximation techniques to cope with the need to represent value functions over large state-action spaces.

Monte Carlo methods

The simplest Monte Carlo methods can be used in an algorithm that mimics policy iteration. Policy iteration consists of two steps: policy evaluation and policy improvement. The Monte Carlo methods are used in the policy evaluation step. In this step, given a stationary, deterministic policy π, the goal is to compute the function values $Q^{\pi}(s,a)$ (or a good approximation to them) for all state-action pairs (s,a). Assume (for simplicity) that the MDP is finite and in fact a table representing the action-values fits into the memory. Further, assume that the problem is episodic and after each episode a new one starts from some random initial state. Then, the estimate of the value of a given state-action pair (s,a)can be computed by simply averaging the sampled returns which originated from (s,a) over time. Given enough time, this procedure can thus construct a precise estimate Q of the action-value function Q^{π}. This finishes the description of the policy evaluation step. In the policy improvement step, as it is done in the standard policy iteration algorithm, the next policy is obtained by computing a greedy policy with respect to Q: Given a state s, this new policy returns an action that maximizes $Q(s,\cdot)$. In

practice one often avoids computing and storing the new policy, but uses lazy evaluation to defer the computation of the maximizing actions to when they are actually needed.

A few problems with this procedure are as follows:

- The procedure may waste too much time on evaluating a suboptimal policy;
- It uses samples inefficiently in that a long trajectory is used to improve the estimate only of the single state-action pair that started the trajectory;
- When the returns along the trajectories have high variance, convergence will be slow;
- It works in episodic problems only;
- It works in small, finite MDPs only.

Temporal difference methods

The first issue is easily corrected by allowing the procedure to change the policy (at all, or at some states) before the values settle. However good this sounds, this may be dangerous as this might prevent convergence. Still, most current algorithms implement this idea, giving rise to the class of generalized policy iteration algorithm. We note in passing that actor critic methods belong to this category.

The second issue can be corrected within the algorithm by allowing trajectories to contribute to any state-action pair in them. This may also help to some extent with the third problem, although a better solution when returns have high variance is to use Sutton's temporal difference (TD) methods which are based on the recursive Bellman equation. Note that the computation in TD methods can be incremental (when after each transition the memory is changed and the transition is thrown away), or batch (when the transitions are collected and then the estimates are computed once based on a large number of transitions). Batch methods, a prime example of which is the least-squares temporal difference method due to Bradtke and Barto (1996), may use the information in the samples better, whereas incremental methods are the only choice when batch methods become infeasible due to their high computational or memory complexity. In addition, there exist methods that try to unify the advantages of the two approaches. Methods based on temporal differences also overcome the second but last issue.

In order to address the last issue mentioned in the previous section, function approximation methods are used. In linear function approximation one starts with a mapping φ that assigns a finite dimensional vector to each state-action pair. Then, the action values of a state-action pair (s,a) are obtained by linearly combining the components of φ(s,a) with some weights θ:

$$Q(s,a) = \sum_{i=1}^{d} \theta_i \phi_i(s,a)$$

.

The algorithms then adjust the weights, instead of adjusting the values associated with the individual state-action pairs. However, linear function approximation is not the only choice. More recently, methods based on ideas from nonparametric statistics (which can be seen to construct their own features) have been explored.

So far, the discussion was restricted to how policy iteration can be used as a basis of the designing reinforcement learning algorithms. Equally importantly, value iteration can also be used as a starting point, giving rise to the Q-Learning algorithm (Watkins 1989) and its many variants.

The problem with methods that use action-values is that they may need highly precise estimates of the competing action values, which can be hard to obtain when the returns are noisy. Though this problem is mitigated to some extent by temporal difference methods and if one uses the so-called compatible function approximation method, more work remains to be done to increase generality and efficiency. Another problem specific to temporal difference methods comes from their reliance on the recursive Bellman equation. Most temporal difference methods have a so-called λ parameter $(0 \leq \lambda \leq 1)$ that allows one to continuously interpolate between Monte-Carlo methods (which do not rely on the Bellman equations) and the basic temporal difference methods (which rely entirely on the Bellman equations), which can thus be effective in palliating this issue.

Direct policy search

An alternative method to find a good policy is to search directly in (some subset) of the policy space, in which case the problem becomes an instance of stochastic optimization. The two approaches available are gradient-based and gradient-free methods.

Gradient-based methods (giving rise to the so-called policy gradient methods) start with a mapping from a finite dimensional (parameter) space to the space of policies: given the parameter vector θ, let π_θ denote the policy associated to θ. Define the performance function by

$$\rho(\theta) = \rho^{\pi_\theta}.$$

Under mild conditions this function will be differentiable as a function of the parameter vector θ. If the gradient of ρ was known, one could use gradient ascent. Since an analytic expression for the gradient is not available, one must rely on a noisy estimate. Such an estimate can be constructed in many ways, giving rise to algorithms like Williams' REINFORCE method (which is also known as the likelihood ratio method in the simulation-based optimization literature). Policy gradient methods have received a lot of attention in the last couple of years (e.g., Peters et al. (2003)), but they remain an active field. The issue with many of these methods is that they may get stuck in local optima (as they are based on local search).

A large class of methods avoids relying on gradient information. These include simulated annealing, cross-entropy search or methods of evolutionary computation. Many gradient-free methods can achieve (in theory and in the limit) a global optimum. In a number of cases they have indeed demonstrated remarkable performance.

The issue with policy search methods is that they may converge slowly if the information based on which they act is noisy. For example, this happens when in episodic problems the trajectories are long and the variance of the returns is large. As argued beforehand, value-function based methods that rely on temporal differences might help in this case. In recent years, several actor-critic algorithms have been proposed following this idea and were demonstrated to perform well on various benchmarks.

Theory

The theory for small, finite MDPs is quite mature. Both the asymptotic and finite-sample behavior of most algorithms is well-understood. As mentioned beforehand, algorithms with provably good online performance (addressing the exploration issue) are known. The theory of large MDPs needs more work. Efficient exploration is largely untouched (except for the case of bandit problems). Although finite-time performance bounds appeared for many algorithms in the recent years, these bounds are expected to be rather loose and thus more work is needed to better understand the relative advantages, as well as the limitations of these algorithms. For incremental algorithm asymptotic convergence issues have been settled. Recently, new incremental, temporal-difference-based algorithms have appeared which converge under a much wider set of conditions than was previously possible (for example, when used with arbitrary, smooth function approximation).

Current research

Current research topics include: adaptive methods which work with fewer (or no) parameters under a large number of conditions, addressing the exploration problem in large MDPs, large scale empirical evaluations, learning and acting under partial information (e.g., using Predictive State Representation), modular and hierarchical reinforcement learning, improving existing value-function and policy search methods, algorithms that work well with large (or continuous) action spaces, transfer learning, lifelong learning, efficient sample-based planning (e.g., based on Monte-Carlo tree search). Multiagent or Distributed Reinforcement Learning is also a topic of interest in current research. There is also a growing interest in real life applications of reinforcement learning. Successes of reinforcement learning are collected on here and here.

Reinforcement learning algorithms such as TD learning are also being investigated as a model for Dopamine-based learning in the brain. In this model, the dopaminergic projections from the substantia nigra to the basal ganglia function as the prediction error. Reinforcement learning has also been used as a part of the model for human skill learning, especially in relation to the interaction between implicit and explicit learning in

skill acquisition (the first publication on this application was in 1995-1996, and there have been many follow-up studies).

Literature

Conferences, journals

Most reinforcement learning papers are published at the major machine learning and AI conferences (ICML, NIPS, AAAI, IJCAI, UAI, AI and Statistics) and journals (JAIR, JMLR, Machine learning journal). Some theory papers are published at COLT and ALT. However, many papers appear in robotics conferences (IROS, ICRA) and the "agent" conference AAMAS. Operations researchers publish their papers at the INFORMS conference and, for example, in the Operation Research, and the Mathematics of Operations Research journals. Control researchers publish their papers at the CDC and ACC conferences, or, e.g., in the journals IEEE Transactions on Automatic Control, or Automatica, although applied works tend to be published in more specialized journals. The Winter Simulation Conference also publishes many relevant papers. Other than this, papers also published in the major conferences of the neural networks, fuzzy, and evolutionary computation communities. The annual IEEE symposium titled Approximate Dynamic Programming and Reinforcement Learning (ADPRL) and the biannual European Workshop on Reinforcement Learning (EWRL) are two regularly held meetings where RL researchers meet.

Unsupervised learning

In machine learning, **unsupervised learning** is a class of problems in which one seeks to determine how the data are organized. Many methods employed here are based on data mining methods used to preprocess data. It is distinguished from supervised learning (and reinforcement learning) in that the learner is given only unlabeled examples.

Unsupervised learning is closely related to the problem of density estimation in statistics. However unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data.

One form of unsupervised learning is clustering. Another example is blind source separation based on Independent Component Analysis (ICA).

Among neural network models, the Self-organizing map (SOM) and Adaptive resonance theory (ART) are commonly used unsupervised learning algorithms. The SOM is a topographic organization in which nearby locations in the map represent inputs with similar properties. The ART model allows the number of clusters to vary with problem size and lets the user control the degree of similarity between members of the same

clusters by means of a user-defined constant called the vigilance parameter. ART networks are also used for many pattern recognition tasks, such as automatic target recognition and seismic signal processing. The first version of ART was "ART1", developed by Carpenter and Grossberg(1988).

Semi-supervised learning

In computer science, **semi-supervised learning** is a class of machine learning techniques that make use of both labeled and unlabeled data for training - typically a small amount of labeled data with a large amount of unlabeled data. Semi-supervised learning falls between unsupervised learning (without any labeled training data) and supervised learning (with completely labeled training data). Many machine-learning researchers have found that unlabeled data, when used in conjunction with a small amount of labeled data, can produce considerable improvement in learning accuracy. The acquisition of labeled data for a learning problem often requires a skilled human agent to manually classify training examples. The cost associated with the labeling process thus may render a fully labeled training set infeasible, whereas acquisition of unlabeled data is relatively inexpensive. In such situations, semi-supervised learning can be of great practical value.

One example of a semi-supervised learning technique is co-training, in which two or possibly more learners are each trained on a set of examples, but with each learner using a different, and ideally independent, set of features for each example.

An alternative approach is to model the joint probability distribution of the features and the labels. For the unlabelled data the labels can then be treated as 'missing data'. Techniques that handle missing data, such as Gibbs sampling or the EM algorithm, can then be used to estimate the parameters of the model.

Transduction (machine learning)

In logic, statistical inference, and supervised learning, **transduction** or **transductive inference** is reasoning from observed, specific (training) cases to specific (test) cases. In contrast, induction is reasoning from observed training cases to general rules, which are then applied to the test cases. The distinction is most interesting in cases where the predictions of the transductive model are not achievable by any inductive model. Note that this is caused by transductive inference on different test sets producing mutually inconsistent predictions.

Transduction was introduced by Vladimir Vapnik in the 1990s, motivated by his view that transduction is preferable to induction since, according to him, induction requires solving a more general problem (inferring a function) before solving a more specific problem (computing outputs for new cases): "When solving a problem of interest, do not

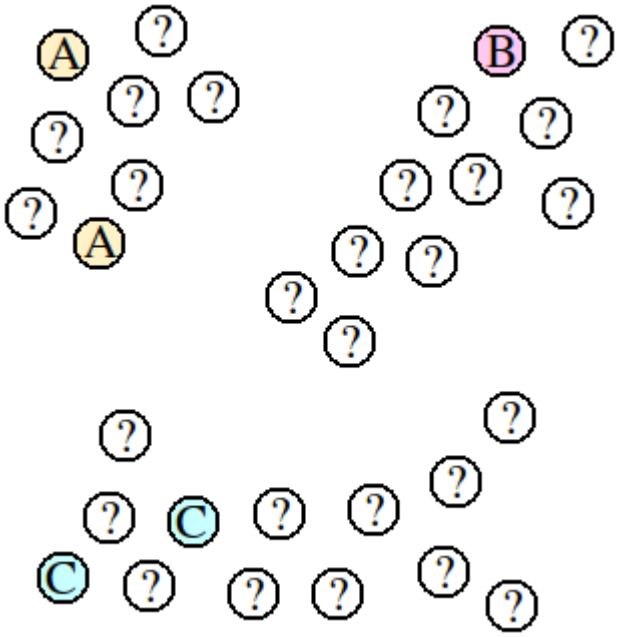
solve a more general problem as an intermediate step. Try to get the answer that you really need but not a more general one."

An example of learning which is not inductive would be in the case of binary classification, where the inputs tend to cluster in two groups. A large set of test inputs may help in finding the clusters, thus providing useful information about the classification labels. The same predictions would not be obtainable from a model which induces a function based only on the training cases. Some people may call this an example of the closely related semi-supervised learning, since Vapnik's motivation is quite different. An example of an algorithm in this category is the Transductive Support Vector Machine (TSVM).

A third possible motivation which leads to transduction arises through the need to approximate. If exact inference is computationally prohibitive, one may at least try to make sure that the approximations are good at the test inputs. In this case, the test inputs could come from an arbitrary distribution (not necessarily related to the distribution of the training inputs), which wouldn't be allowed in semi-supervised learning. An example of an algorithm falling in this category is the Bayesian Committee Machine (BCM).

Example Problem

The following example problem contrasts some of the unique properties of transduction against induction.



A collection of points is given, such that some of the points are labeled (A, B, or C), but most of the points are unlabeled (?). The goal is to predict appropriate labels for all of the unlabeled points.

The inductive approach to solving this problem is to use the labeled points to train a supervised learning algorithm, and then have it predict labels for all of the unlabeled points. With this problem, however, the supervised learning algorithm will only have five labeled points to use as a basis for building a predictive model. It will certainly struggle to build a model that captures the structure of this data. For example, if a nearest-neighbor algorithm is used, then the points near the middle will be labeled "A" or "C", even though it is apparent that they belong to the same cluster as the point labeled "B".

Transduction has the advantage of being able to consider all of the points, not just the labeled points, while performing the labeling task. In this case, transductive algorithms would label the unlabeled points according to the clusters to which they naturally belong. The points in the middle, therefore, would most likely be labeled "B", because they are packed very close to that cluster.

An advantage of transduction is that it may be able to make better predictions with fewer labeled points, because it uses the natural breaks found in the unlabeled points. One disadvantage of transduction is that it builds no predictive model. If a previously unknown point is added to the set, the entire transductive algorithm would need to be repeated with all of the points in order to predict a label. This can be computationally expensive if the data is made available incrementally in a stream. Further, this might cause the predictions of some of the old points to change (which may be good or bad, depending on the application). A supervised learning algorithm, on the other hand, can label new points instantly, with very little computational cost.

Transduction Algorithms

Transduction algorithms can be broadly divided into two categories: 1- Those that seek to assign discrete labels to unlabeled points, and 2- Those that seek to regress continuous labels for unlabeled points. Algorithms that seek to predict discrete labels tend to be derived by adding partial-supervision to a clustering algorithm. These can be further sub-divided into two categories: those that cluster by partitioning, and those that cluster by agglomerating. Algorithms that seek to predict continuous labels tend to be derived by adding partial-supervision to a manifold learning algorithm.

Partitioning Transduction

Partitioning-transduction can be thought of as top-down transduction. It is a semi-supervised extension of partition-based clustering. It is typically performed as follows:

```
Consider the set of all points to be one large partition.
while any partition P contains two points with conflicting labels:
  Partition P into smaller partitions.
for each partition P:
  Assign the same label to all of the points in P.
```

Of course, any reasonable partitioning technique could be used with this algorithm. Max flow min cut partitioning schemes are very popular for this purpose.

Agglomerative Transduction

Agglomerative transduction can be thought of as bottom-up transduction. It is a semi-supervised extension of agglomerative clustering. It is typically performed as follows:

```
Compute the pair-wise distances, D, between all the points.
Sort D in ascending order.
Consider each point to be a cluster of size 1.
for each pair of points {a,b} in D:
  if (a is unlabeled) or (b is unlabeled) or (a and b have the same
label)
    Merge the two clusters that contain a and b.
    Label all points in the merged cluster with the same label.
```

Manifold Transduction

Manifold-learning-based transduction is still a very young field of research.

Multi-task learning

Multi-task learning is an approach to machine learning, that learns a problem together with other related problems at the same time, using a shared representation. This often leads to a better model for the main task, because it allows the learner to use the commonality among the tasks. Therefore, multi-task learning is a kind of inductive transfer.

Chapter 18

Computational Learning Theory

In theoretical computer science, **computational learning theory** is a mathematical field related to the analysis of machine learning algorithms.

Overview

Theoretical results in machine learning mainly deal with a type of inductive learning called supervised learning. In supervised learning, an algorithm is given samples that are labeled in some useful way. For example, the samples might be descriptions of mushrooms, and the labels could be whether or not the mushrooms are edible. The algorithm takes these previously labeled samples and uses them to induce a classifier. This classifier is a function that assigns labels to samples including the samples that have never been previously seen by the algorithm. The goal of the supervised learning algorithm is to optimize some measure of performance such as minimizing the number of mistakes made on new samples.

In addition to performance bounds, computational learning theorists study the time complexity and feasibility of learning. In computational learning theory, a computation is considered feasible if it can be done in polynomial time. There are two kinds of time complexity results:

- Positive results – Showing that a certain class of functions is learnable in polynomial time.
- Negative results – Showing that certain classes cannot be learned in polynomial time.

Negative results are proven only by assumption. The assumptions that are common in negative results are:

- Computational complexity - $P \neq NP$
- Cryptographic - One-way functions exist.

There are several different approaches to computational learning theory. These differences are based on making assumptions about the inference principles used to generalize from limited data. This includes different definitions of probability and different assumptions on the generation of samples. The different approaches include:

Probably approximately correct learning

In computational learning theory, **probably approximately correct learning** (**PAC learning**) is a framework for mathematical analysis of machine learning. It was proposed in 1984 by Leslie Valiant.

In this framework, the learner receives samples and must select a generalization function (called the hypothesis) from a certain class of possible functions. The goal is that, with high probability (the "probably" part), the selected function will have low generalization error (the "approximately correct" part). The learner must be able to learn the concept given any arbitrary approximation ratio, probability of success, or distribution of the samples.

The model was later extended to treat noise (misclassified samples).

An important innovation of the PAC framework is the introduction of computational complexity theory concepts to machine learning. In particular, the learner is expected to find efficient functions (time and space requirements bounded to a polynomial of the example size), and the learner itself must implement an efficient procedure (requiring an example count bounded to a polynomial of the concept size, modified by the approximation and likelihood bounds).

Definitions and terminology

In order to give the definition for something that is PAC-learnable, we first have to introduce some terminology.

For the following definitions, two examples will be used. The first is the problem of character recognition given an array of n bits. The other example is the problem of finding an interval that will correctly classify points within the interval as positive and the points outside of the range as negative.

Let X be a set called the instance space or the encoding of all the samples. In the character recognition problem, the instance space is $X = \{0,1\}^n$. In the interval problem the instance space is $X = \mathbb{R}$, where $\mathbb{R}$ denotes the set of all real numbers.

A concept is a subset $c \subset X$. One concept is the set of all of the bits that encode for the letter "P" in X = {0,1}^{n}. An example concept from the second example is the set of all of the numbers between π / 2 and $\sqrt{10}$. A concept class is a set of concepts over X. This could be the set of all of the array of bits that are skeletonized 4-connected (width of the font is 1).

Let EX(c,D) be a procedure that draws an example, x, using a probability distribution D and gives the correct label c(x).

Say that there is an algorithm A that given access to EX(c,D) and inputs ε and δ that, with probability of at least 1 − δ, A outputs a hypothesis $h \in C$ that has error less than or equal to ε with examples drawn from X with the distribution D. If there is such an algorithm for every concept $c \in C$, for every distribution D over X, and for all 0 < ε < 1 / 2 and 0 < δ < 1 / 2 then C is **PAC learnable**. We can also say that A is a **PAC learning algorithm** for C.

Vapnik–Chervonenkis theory

Vapnik–Chervonenkis theory (also known as **VC theory**) was developed during 1960–1990 by Vladimir Vapnik and Alexey Chervonenkis. The theory is a form of computational learning theory, which attempts to explain the learning process from a statistical point of view.

VC theory is related to **statistical learning theory** and to empirical processes. Richard M. Dudley and Vladimir Vapnik himself, among others, apply VC-theory to empirical processes.

VC theory covers at least four parts (as explained in The Nature of Statistical Learning Theory):

- Theory of consistency of learning processes
  - What are (necessary and sufficient) conditions for consistency of a learning process based on the empirical risk minimization principle ?
- Nonasymptotic theory of the rate of convergence of learning processes
  - How fast is the rate of convergence of the learning process?
- Theory of controlling the generalization ability of learning processes
  - How can one control the rate of convergence (the generalization ability) of the learning process?
- Theory of constructing learning machines
  - How can one construct algorithms that can control the generalization ability?

In addition, VC theory and VC dimension are instrumental in the theory of empirical processes, in the case of processes indexed by VC classes.

The last part of VC theory introduced a well-known learning algorithm: the support vector machine.

VC theory contains important concepts such as the VC dimension and structural risk minimization. This theory is related to mathematical subjects such as:

- reproducing kernel Hilbert spaces
- regularization networks
- kernels
- empirical processes

Algorithmic learning theory

Algorithmic learning theory (or **algorithmic inductive inference**) is a framework for machine learning.

The framework was introduced in E. Mark Gold's seminal paper "Language identification in the limit". The objective of language identification is for a machine running one program to be capable of developing another program by which any given sentence can be tested to determine whether it is "grammatical" or "ungrammatical". The language being learned need not be English or any other natural language - in fact the definition of "grammatical" can be absolutely anything known to the tester.

In the framework of algorithmic learning theory, the tester gives the learner an example sentence at each step, and the learner responds with a hypothesis, which is a suggested program to determine grammatical correctness. It is required of the tester that every possible sentence (grammatical or not) appears in the list eventually, but no particular order is required. It is required of the learner that at each step the hypothesis must be correct for all the sentences so far.

A particular learner is said to be able to "learn a language in the limit" if there is a certain number of steps beyond which its hypothesis no longer changes. At this point it has indeed learned the language, because every possible sentence appears somewhere in the sequence of inputs (past or future), and the hypothesis is correct for all inputs (past or future), so the hypothesis is correct for every sentence. The learner is not required to be able to tell when it has reached a correct hypothesis, all that is required is that it be true.

Gold showed that any language which is defined by a Turing machine program can be learned in the limit by another Turing-complete machine using enumeration. This is done

by the learner testing all possible Turing machine programs in turn until one is found which is correct so far - this forms the hypothesis for the current step. Eventually, the correct program will be reached, after which the hypothesis will never change again (but note that the learner does not know that it won't need to change).

Gold also showed that if the learner is given only positive examples (that is, only grammatical sentences appear in the input, not ungrammatical sentences), then the language can only be guaranteed to be learned in the limit if there are only a finite number of possible sentences in the language (this is possible if, for example, sentences are known to be of limited length).

Language identification in the limit is a very theoretical model. It does not allow for limits of runtime or computer memory which can occur in practice, and the enumeration method may fail if there are errors in the input. However the framework is very powerful, because if these strict conditions are maintained, it allows the learning of any program known to be computable. This is because a Turing machine program can be written to mimic any program in any conventional programming language.

Other frameworks of learning consider a much more restricted class of function than Turing machines, but complete the learning more quickly (in polynomial time). An example of such a framework is Probably approximately correct learning.

Online machine learning

In machine learning, **online learning** is a model of induction that learns one instance at a time. The goal in online learning is to predict labels for instances. For example, the instances could describe the current conditions of the stock market, and an online algorithm predicts tomorrow's value of a particular stock. The key defining characteristic of online learning is that soon after the prediction is made, the true label of the instance is discovered. This information can then be used to refine the prediction hypothesis used by the algorithm. The goal of the algorithm is to make predictions that are close to the true labels.

More formally, an online algorithm proceeds in a sequence of trials. Each trial can be decomposed into three steps. First the algorithm receives an instance. Second the algorithm predicts the label of the instance. Third the algorithm receives the true label of the instance. The third stage is the most crucial as the algorithm can use this label feedback to update its hypothesis for future trials. The goal of the algorithm is to minimize some performance criteria. For example, with stock market prediction the algorithm may attempt to minimize sum of the square distances between the predicted and true value of a stock. Another popular performance criteria is to minimize the number of mistakes when dealing with classification problems.

Because online learning algorithms continually receive label feedback, the algorithms are able to adapt and learn in difficult situations. Many online algorithms can give strong guarantees on performance even when the instances are not generated by a distribution. As long a reasonably good classifier exists, the online algorithm will learn to predict correct labels. This good classifier must come from a previously determined set that depends on the algorithm. For example, two popular on-line algorithms perceptron and winnow can perform well when a hyperplane exists that splits the data into two categories. These algorithms can even be modified to do provably well even if the hyperplane is allowed to infrequently change during the online learning trials.

Unfortunately, the main difficulty of online learning is also a result of the requirement for continual label feedback. For many problems it is not possible to guarantee that accurate label feedback will be available in the near future. For example, when designing a system that learns how to do optical character recognition, typically some expert must label previous instances to help train the algorithm. In actual use of the OCR application, the expert is no longer available and no inexpensive outside source of accurate labels is available. Fortunately, there is a large class of problems where label feedback is always available. For any problem that consists of predicting the future, an online learning algorithm just needs to wait for the label to become available. This is true in our previous example of stock market prediction and many other problems.

Computational learning theory has led to several practical algorithms. For example, PAC theory inspired boosting, VC theory led to support vector machines, and Bayesian inference led to belief networks (by Judea Pearl).

Chapter 19

Support Vector Machine

Support vector machines (**SVMs**) are a set of related supervised learning methods that analyze data and recognize patterns, used for classification and regression analysis. The original SVM algorithm was invented by Vladimir Vapnik and the current standard incarnation (soft margin) was proposed by Corinna Cortes and Vladimir Vapnik . The standard SVM takes a set of input data, and predicts, for each given input, which of two possible classes the input is a member of, which makes the SVM a non-probabilistic binary linear classifier. Since an SVM is a classifier, then given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other. Intuitively, an SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

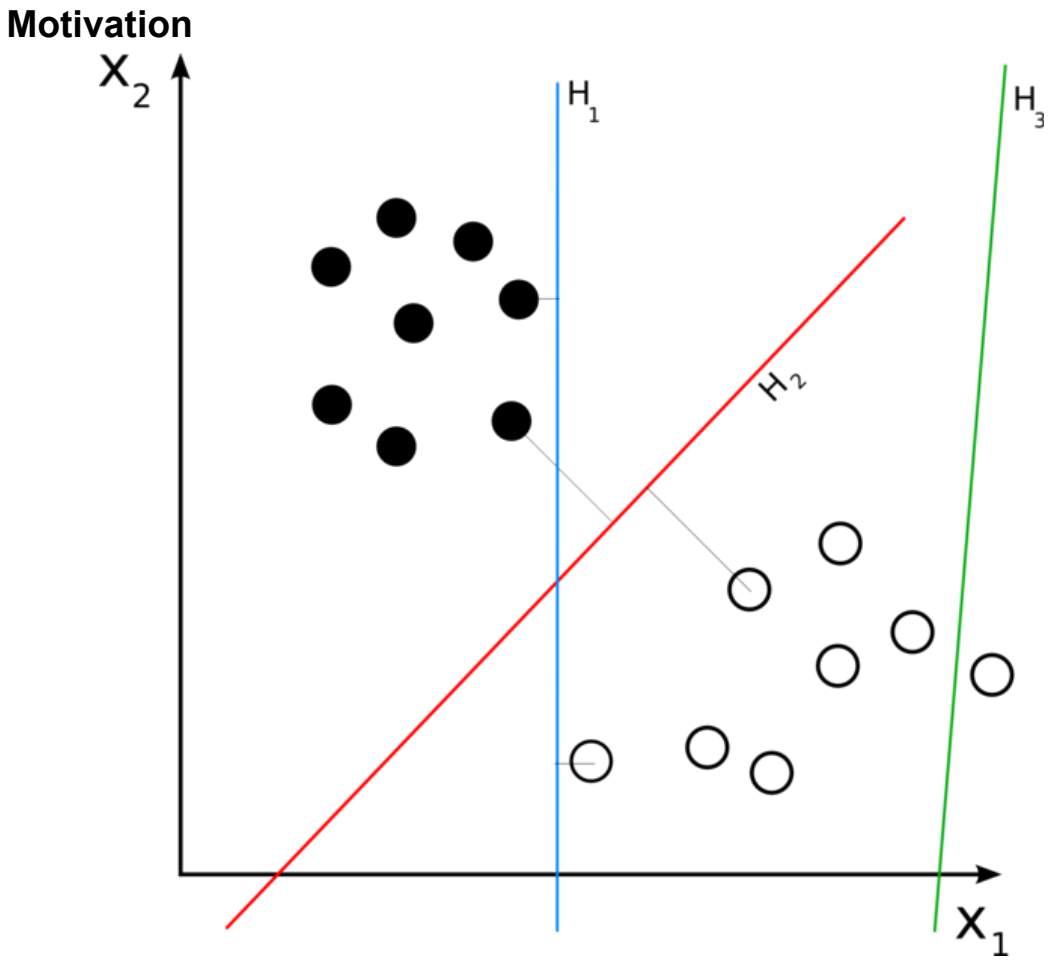
More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Whereas the original problem may be stated in a finite dimensional space, it often happens that in that space the sets to be discriminated are not linearly separable. For this reason it was proposed that the original finite dimensional space be mapped into a much higher dimensional space presumably making the separation easier in that space. SVM schemes use a mapping into a larger space so that cross products may be computed easily in terms of the variables in the original space making the computational load reasonable. The cross products in the larger space are defined in terms of a kernel function K(x,y) which can be selected to suit the problem. The hyperplanes in the large space are defined as the set of points whose cross product with a vector in that space is constant. The vectors defining the hyperplanes can be chosen to be linear combinations with parameters

α_i of images of feature vectors which occur in the data base. With this choice of a hyperplane the points x in the feature space which are mapped into the hyperplane are defined by the relation:

$$\sum_i \alpha_i K(x_i, x) = \text{constant}$$

Note that if K(x,y) becomes small as y grows further from x, each element in the sum measures the degree of closeness of the test point x to the corresponding data base point x_i. In this way the sum of kernels above can be used to measure the relative nearness of each test point to the data points originating in one or the other of the sets to be discriminated. Note the fact that the set of points x mapped into any hyperplane can be quite convoluted as a result allowing much more complex discrimination between sets which are far from convex in the original space.



H3 (green) doesn't separate the 2 classes. H1 (blue) does, with a small margin and H2 (red) with the maximum margin.

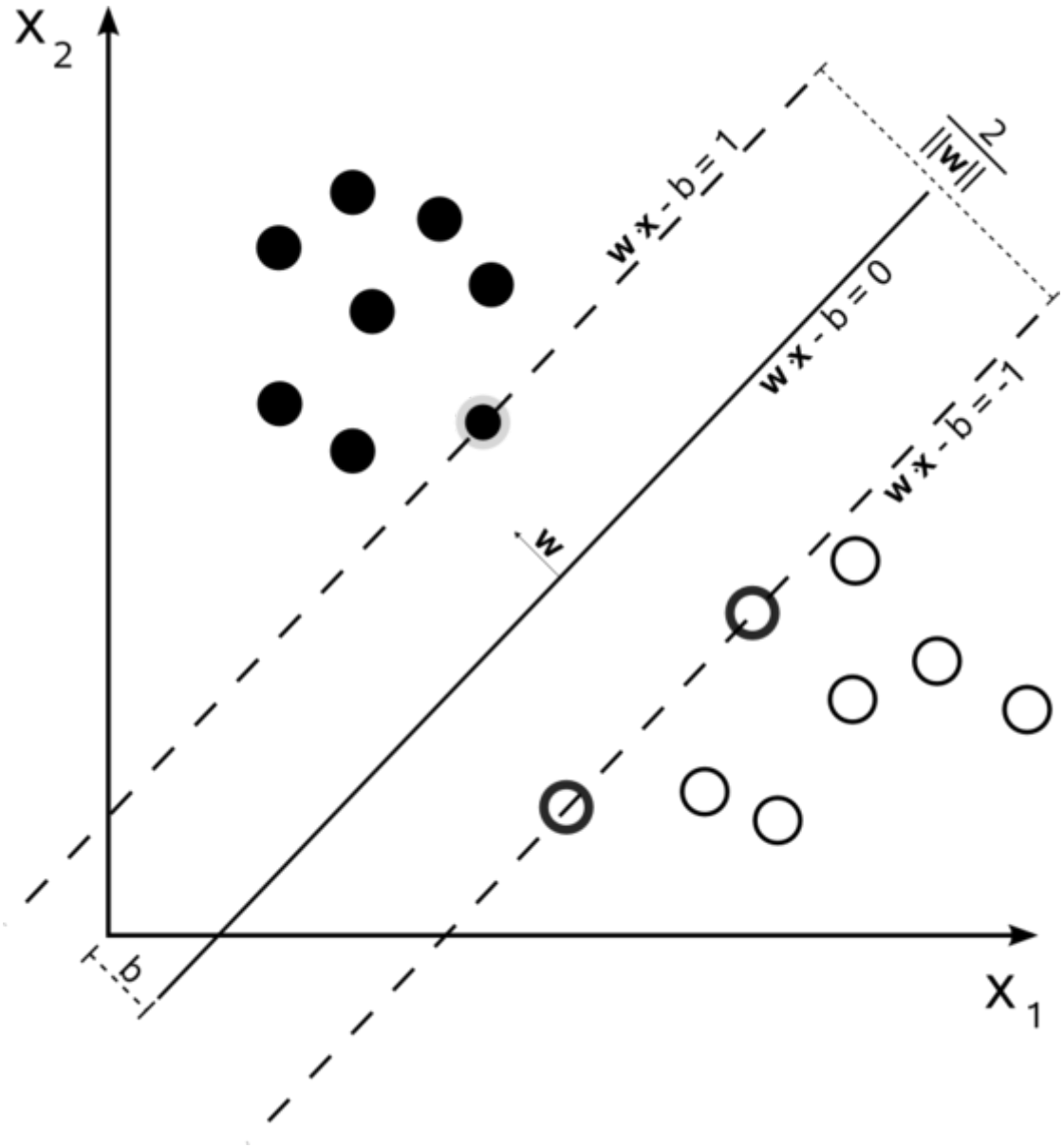
Classifying data is a common task in machine learning. Suppose some given data points each belong to one of two classes, and the goal is to decide which class a new data point will be in. In the case of support vector machines, a data point is viewed as a p-dimensional vector (a list of p numbers), and we want to know whether we can separate such points with a p − 1-dimensional hyperplane. This is called a linear classifier. There are many hyperplanes that might classify the data. One reasonable choice as the best hyperplane is the one that represents the largest separation, or margin, between the two classes. So we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum margin classifier.

Formalization

We are given some training data $\mathcal{D}$, a set of n points of the form

$$\mathcal{D} = \{(\mathbf{x}_i, c_i) \mid \mathbf{x}_i \in \mathbb{R}^p,\ c_i \in \{-1, 1\}\}_{i=1}^{n}$$

where the c_i is either 1 or −1, indicating the class to which the point $\mathbf{x}_i$ belongs. Each $\mathbf{x}_i$ is a p-dimensional real vector. We want to find the maximum-margin hyperplane that divides the points having $c_i = 1$ from those having $c_i = -1$. Any hyperplane can be written as the set of points $\mathbf{x}$ satisfying



Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.

$$\mathbf{w} \cdot \mathbf{x} - b = 0,$$

where $\cdot$ denotes the dot product. The vector $\mathbf{w}$ is a normal vector: it is perpendicular to the hyperplane. The parameter $\frac{b}{\|\mathbf{w}\|}$ determines the offset of the hyperplane from the origin along the normal vector $\mathbf{w}$.

We want to choose the $\mathbf{w}$ and b to maximize the margin, or distance between the parallel hyperplanes that are as far apart as possible while still separating the data. These hyperplanes can be described by the equations

$$\mathbf{w} \cdot \mathbf{x} - b = 1$$

and

$$\mathbf{w} \cdot \mathbf{x} - b = -1.$$

Note that if the training data are linearly separable, we can select the two hyperplanes of the margin in a way that there are no points between them and then try to maximize their distance. By using geometry, we find the distance between these two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, so we want to minimize $\|\mathbf{w}\|$. As we also have to prevent data points falling into the margin, we add the following constraint: for each i either

$$\mathbf{w} \cdot \mathbf{x}_i - b \geq 1 \qquad \text{for } \mathbf{x}_i \text{of the first class}$$

or

$$\mathbf{w} \cdot \mathbf{x}_i - b \leq -1 \qquad \text{for } \mathbf{x}_i \text{of the second.}$$

This can be rewritten as:

$$c_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1, \quad \text{for all } 1 \leq i \leq n. \qquad (1)$$

We can put this together to get the optimization problem:

Minimize (in $\mathbf{w}, b$)

$$\|\mathbf{w}\|$$

subject to (for any $i = 1, \ldots, n$)

$$c_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1.$$

Primal form

The optimization problem presented in the preceding section is difficult to solve because it depends on ||**w**||, the norm of **w**, which involves a square root. Fortunately it is possible to alter the equation by substituting ||**w**|| with $\frac{1}{2}\|\mathbf{w}\|^2$ (the factor of 1/2 being used for mathematical convenience) without changing the solution (the minimum of the original and the modified equation have the same **w** and b). This is a quadratic programming (QP) optimization problem. More clearly:

Minimize (in $\mathbf{w}, b$)

$$\frac{1}{2}\|\mathbf{w}\|^2$$

subject to (for any $i = 1, \ldots, n$)

$$c_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1.$$

One could be tempted to express the previous problem by means of non-negative Lagrange multipliers α_i as

$$\min_{\mathbf{w},b,\boldsymbol{\alpha}} \{\frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i [c_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1]\}$$

but this would be wrong. The reason is the following: suppose we can find a family of hyperplanes which divide the points; then all $c_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1 \geq 0$. Hence we could find the minimum by sending all α_i to $+\infty$, and this minimum would be reached for all the members of the family, not only for the best one which can be chosen solving the original problem.

Nevertheless the previous constrained problem can be expressed as

$$\min_{\mathbf{w},b} \max_{\boldsymbol{\alpha}} \{\frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i [c_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1]\}$$

that is we look for a saddle point. In doing so all the points which can be separated as $c_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1 > 0$ do not matter since we must set the corresponding α_i to zero.

This problem can now be solved by standard quadratic programming techniques and programs. The solution can be expressed by terms of linear combination of the training vectors as

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i c_i \mathbf{x_i}$$

Only a few α_i will be greater than zero. The corresponding $\mathbf{x_i}$ are exactly the support vectors, which lie on the margin and satisfy $c_i(\mathbf{w} \cdot \mathbf{x_i} - b) = 1$. From this one can derive that the support vectors also satisfy

$$\mathbf{w} \cdot \mathbf{x_i} - b = 1/c_i = c_i \iff b = \mathbf{w} \cdot \mathbf{x_i} - c_i$$

which allows one to define the offset b. In practice, it is more robust to average over all N_{SV} support vectors:

$$b = \frac{1}{N_{SV}} \sum_{i=1}^{N_{SV}} (\mathbf{w} \cdot \mathbf{x_i} - c_i)$$

Dual form

Writing the classification rule in its unconstrained dual form reveals that the maximum margin hyperplane and therefore the classification task is only a function of the support vectors, the training data that lie on the margin.

Using the fact, that $\|\mathbf{w}\|^2 = w \cdot w$ and substituting $\mathbf{w} = \sum_{i=1}^{n} \alpha_i c_i \mathbf{x_i}$, one can show that the dual of the SVM reduces to the following optimization problem:

Maximize (in α_i)

$$\tilde{L}(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j c_i c_j \mathbf{x}_i^T \mathbf{x}_j = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j c_i c_j k(\mathbf{x}_i, \mathbf{x}_j)$$

subject to (for any $i = 1, \ldots, n$)

$$\alpha_i \geq 0,$$

and to the constraint from the minimization in b

$$\sum_{i=1}^{n} \alpha_i c_i = 0.$$

Here the kernel is defined by $k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$.

The α terms constitute a dual representation for the weight vector in terms of the training set:

$$\mathbf{w} = \sum_i \alpha_i c_i \mathbf{x}_i.$$

Biased and unbiased hyperplanes

For simplicity reasons, sometimes it is required that the hyperplane passes through the origin of the coordinate system. Such hyperplanes are called unbiased, whereas general hyperplanes not necessarily passing through the origin are called biased. An unbiased

hyperplane can be enforced by setting b = 0 in the primal optimization problem. The corresponding dual is identical to the dual given above without the equality constraint

$$\sum_{i=1}^{n} \alpha_i c_i = 0.$$

Transductive support vector machines

Transductive support vector machines extend SVMs in that they could also treat partially labeled data in semi-supervised learning. Here, in addition to the training set $\mathcal{D}$, the learner is also given a set

$$\mathcal{D}^\star = \{\mathbf{x}_i^\star | \mathbf{x}_i^\star \in \mathbb{R}^p\}_{i=1}^k$$

of test examples to be classified. Formally, a transductive support vector machine is defined by the following primal optimization problem:

Minimize (in $\mathbf{w}, b, \mathbf{c}^\star$)

$$\frac{1}{2}\|\mathbf{w}\|^2$$

subject to (for any $i = 1, \ldots, n$ and any $j = 1, \ldots, k$)

$$c_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1,$$
$$c_j^\star(\mathbf{w} \cdot \mathbf{x_j^\star} - b) \geq 1,$$

and

$$c_j^\star \in \{-1, 1\}.$$

Transductive support vector machines were introduced by Vladimir Vapnik in 1998.

Properties

SVMs belong to a family of generalized linear classifiers. They can also be considered a special case of Tikhonov regularization. A special property is that they simultaneously minimize the empirical classification error and maximize the geometric margin; hence they are also known as **maximum margin classifiers**.

A comparison of the SVM to other classifiers has been made by Meyer, Leisch and Hornik.

Extensions to the linear SVM

Soft margin

In 1995, Corinna Cortes and Vladimir Vapnik suggested a modified maximum margin idea that allows for mislabeled examples. If there exists no hyperplane that can split the "yes" and "no" examples, the Soft Margin method will choose a hyperplane that splits the examples as cleanly as possible, while still maximizing the distance to the nearest cleanly split examples. The method introduces slack variables, ξ_i, which measure the degree of misclassification of the datum x_i

$$c_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1 - \xi_i \quad 1 \leq i \leq n. \qquad (2)$$

The objective function is then increased by a function which penalizes non-zero ξ_i, and the optimization becomes a trade off between a large margin, and a small error penalty. If the penalty function is linear, the optimization problem becomes:

$$\min_{\mathbf{w},\xi} \left\{ \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i \right\}$$

subject to (for any $i = 1, \ldots n$)

$$c_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1 - \xi_i, \qquad \xi_i \geq 0.$$

This constraint in (2) along with the objective of minimizing $\|\mathbf{w}\|$ can be solved using Lagrange multipliers as done above. One has then to solve the following problem

$$\min_{\mathbf{w},\xi,b} \max_{\alpha,\beta} \left\{ \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i[c_i(\mathbf{w} \cdot \mathbf{x_i} - b) - 1 + \xi_i] - \sum_{i=1}^{n} \beta_i \xi_i \right\}$$

with $\alpha_i, \beta_i \geq 0$.

The key advantage of a linear penalty function is that the slack variables vanish from the dual problem, with the constant C appearing only as an additional constraint on the Lagrange multipliers. For the above formulation and its huge impact in practice, Cortes and Vapnik received the 2008 ACM Paris Kanellakis Award . Nonlinear penalty functions have been used, particularly to reduce the effect of outliers on the classifier, but unless care is taken, the problem becomes non-convex, and thus it is considerably more difficult to find a global solution.

Nonlinear classification

The original optimal hyperplane algorithm proposed by Vladimir Vapnik in 1963 was a linear classifier. However, in 1992, Bernhard Boser, Isabelle Guyon and Vapnik suggested a way to create nonlinear classifiers by applying the kernel trick (originally proposed by Aizerman et al.) to maximum-margin hyperplanes. The resulting algorithm is formally similar, except that every dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The transformation may be nonlinear and the transformed space high dimensional; thus though the classifier is a hyperplane in the high-dimensional feature space, it may be nonlinear in the original input space.

If the kernel used is a Gaussian radial basis function, the corresponding feature space is a Hilbert space of infinite dimension. Maximum margin classifiers are well regularized, so the infinite dimension does not spoil the results. Some common kernels include:

- Polynomial (homogeneous): $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j})^d$
- Polynomial (inhomogeneous): $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$
- Gaussian or Radial Basis Function: $k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma\|\mathbf{x_i} - \mathbf{x_j}\|^2)$, for $\gamma > 0$. Sometimes parametrized using $\gamma = 1 / 2\sigma^2$
- Hyperbolic tangent: $k(\mathbf{x_i}, \mathbf{x_j}) = \tanh(\kappa\mathbf{x_i} \cdot \mathbf{x_j} + c)$, for some (not every) $\kappa > 0$ and $c < 0$

The kernel is related to the transform $\varphi(\mathbf{x_i})$ by the equation $k(\mathbf{x_i}, \mathbf{x_j}) = \varphi(\mathbf{x_i}) \cdot \varphi(\mathbf{x_j})$. The value **w** is also in the transformed space, with $\mathbf{w} = \sum_i \alpha_i c_i \varphi(\mathbf{x}_i)$. Dot products with **w** for classification can again be computed by the kernel trick, i.e. $\mathbf{w} \cdot \varphi(\mathbf{x}) = \sum_i \alpha_i c_i k(\mathbf{x}_i, \mathbf{x})$. However, there does not in general exist a value **w'** such that $\mathbf{w} \cdot \varphi(\mathbf{x}) = k(\mathbf{w}', \mathbf{x})$.

Parameter selection

The effectiveness of SVM depends on the selection of kernel, the kernel's parameters, and soft margin parameter C.

A common choice is a Gaussian kernel, which has a single parameter γ. Best combination of C and γ is often selected by a grid-search with exponentially growing sequences of C and γ, for example, $C \in \{2^{-5}, 2^{-3}, \ldots, 2^{13}, 2^{15}\}$; $\gamma \in \{2^{-15}, 2^{-13}, \ldots, 2^{1}, 2^{3}\}$. Typically, each combination of parameter choices is checked using cross validation, and the parameters with best cross-validation accuracy are picked. The final model, which is used for testing and for classifying new data, is then trained on the whole training set using the selected parameters.

Issues

Potential drawbacks of the SVM are the following two aspects:

- Uncalibrated Class membership probabilities
- The SVM is only directly applicable for two-class tasks. Therefore, algorithms that reduce the multi-class task to several binary problems have to be applied.

Multiclass SVM

Multiclass SVM aims to assign labels to instances by using support vector machines, where the labels are drawn from a finite set of several elements. The dominating approach for doing so is to reduce the single multiclass problem into multiple binary classification problems. Each of the problems yields a binary classifier, which is assumed to produce an output function that gives relatively large values for examples from the positive class and relatively small values for examples belonging to the negative class. Two common methods to build such binary classifiers are where each classifier distinguishes between (i) one of the labels to the rest (one-versus-all) or (ii) between every pair of classes (one-versus-one). Classification of new instances for one-versus-all case is done by a winner-takes-all strategy, in which the classifier with the highest output function assigns the class (it is important that the output functions be calibrated to produce comparable scores). For the one-versus-one approach, classification is done by a max-wins voting strategy, in which every classifier assigns the instance to one of the two classes, then the vote for the assigned class is increased by one vote, and finally the class with most votes determines the instance classification.

Structured SVM

SVMs have been generalized to structured SVMs, where the label space is structured and of possibly infinite size.

Regression

A version of SVM for regression was proposed in 1996 by Vladimir Vapnik, Harris Drucker, Chris Burges, Linda Kaufman and Alex Smola. This method is called support vector regression (SVR). The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by SVR depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction (within a threshold ε). There is also a least squares version of support vector machine (SVM) called least squares support vector machine (LS-SVM) proposed in Suykens and Vandewalle.

Implementation

The parameters of the maximum-margin hyperplane are derived by solving the optimization. There exist several specialized algorithms for quickly solving the QP problem that arises from SVMs, mostly reliant on heuristics for breaking the problem down into smaller, more-manageable chunks. A common method for solving the QP problem is Platt's Sequential Minimal Optimization (SMO) algorithm, which breaks the problem down into 2-dimensional sub-problems that may be solved analytically, eliminating the need for a numerical optimization algorithm.

Another approach is to use an interior point method that uses Newton-like iterations to find a solution of the Karush-Kuhn-Tucker conditions of the primal and dual problems. Instead of solving a sequence of broken down problems, this approach directly solves the problem as a whole. To avoid solving a linear system involving the large kernel matrix, a low rank approximation to the matrix is often used to use the kernel trick.