CSCI 4150: Introduction to Artificial Intelligence

Thomas P. Trappenberg
Dalhousie University

2009

Acknowledgements

These lecture notes are mostly based on lecture nodes by Andrew Ng from Stanford University. Specifically, we follow basically his lecture notes of 'CS221: Artificial Intelligence: Principles and Techniques' from winter 2009, and some notes from his course 'CS229: Machine Learning' from fall 2008. The full set of these lecture notes are available on his web site (`http://robotics.stanford.edu/~ang`). Videos of Andrew's lectures on machine learning are also available on his web site, and these are very much recommended for further studies.

Andrew's material is augmented with some material from slides by Stuart Russel and Peter Norvig for their book *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice Hall, 2003. This book is recommended reading for CSCI 4150. Some additional material was provided by Evangelos Millios from earlier offerings of this course.

Contents

Introduction and History

1

A precise definition of artificial intelligence (AI) is not easy and often controversial. This introductory chapter outlines some areas associated with AI in a historical context and by highlighting some modern approaches in AI. Many problems in AI can be formulated as search problems, which we review in the first part of the course. The course then concentrates on machine learning (ML) and probabilistic reasoning.

1.1 Brief history of AI

1943	McCulloch & Pitts: Boolean circuit model of brain
1950	Turing's "Computing Machinery and Intelligence"
1952–69	Look, Ma, no hands!
1950s	Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
1956	Dartmouth meeting: "Artificial Intelligence" adopted
1965	Robinson's complete algorithm for logical reasoning
1966–74	AI discovers computational complexity Neural network research almost disappears
1969–79	Early development of knowledge-based systems
1980–88	Expert systems industry booms
1988–93	Expert systems industry busts: "AI Winter"
1985–95	Neural networks return to popularity
1988–	Resurgence of probability; general increase in technical depth "Nouvelle AI": ALife, GAs, soft computing
1995–	Agents, agents, everywhere . . . Machine learning comes to age, web intelligence, smart machines
2003–	Human-level AI back on the agenda

The term AI was born in 1956, at a workshop in Dartmouth organized by John McCarthy. Those gathered agreed to adopt McCarthys name for the new field: Artificial Intelligence. At that point, there was lots of enthusiasm. Things seemed to work out really well. Only a few years before, computers were viewed as large calcula- tors, and now truly intelligent systems seemed within reach. Early programs did amazing things by simply representing knowledge about a domain and searching for a solution. For example, Newell & Simon's 'Logic Theorist' proved qualitative mathematical theorems, and even found a shorter proof for one of the theorems in Russell and Whitehead's 'Principia Mathematica'. In 1958, McCarthy suggested how the same paradigm could be used for commonsense reasoning: represent knowledge about the everyday world as logical axioms, and use that knowledge to figure out how to act. Amazingly, a general-purpose logical theorem prover was able (for instance) to generate a plan for driving to the airport. Arguably the first convincing machine learning program, Arthur Samuel's Checkers playing program started out playing poorly, but learned to play better by playing many games against itself. Growing to play better than Samuel, this program disproved the (still-made) argument that computers can only do what they are told to do. A particularly good example of how a simple set of rules can produce seemingly complex behavior was Joseph Weizenbaum's Eliza program, which simulates a Rogerian psychotherapist. Although Eliza's algorithms are best described as simple pattern matching, and it was not intended as a serious attempt at machine intelligence, it still produced appropriate responses to a variety of statements. Because of programs like Eliza, there was also a hope of building systems in the near future that would pass the Turing Test for machine intelligence. In the Turing Test, a human judge sits at a computer terminal, and chats via instant messenger with one of two entities: either a human or an AI computer program. The human and computer would each try to convince the human judge that they are the human. If the judge is unable tell whether she is chatting with a human or the AI program, then the AI program passes the Turing Test. (Turing considered this a sufficient, but not necessary, condition for intelligence, since a machine could be intelligent without being able to impersonate a human.) Things seemed very rosy. Herb Simon, in 1957, said:

> *It is not my aim to surprise or shock you – but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until – in a visible future – the range of problems they can handle will be coextensive with the range to which human mind has been applied.*
> *More precisely: within 10 years a computer would be chess champion, and an important new mathematical theorem would be proved by a computer.*

Both of these milestones have now been achieved by computers, but each took closer to 40 years, rather than 10. After the initial enthusiasm, there was the dawning realization that problems are much harder than one originally thought, and that simple tricks dont work. For instance, one of Elizas rules was that if the user utters the word 'mother', then respond 'Tell me more about your family'. This sometimes works well, but it can also generate some very unnatural responses. For example, if you say 'I wanted to adopt a puppy, but its too young to be separated from its mother', Eliza may also respond 'Tell me more about your family'. Another example was machine translation. Much time and money were spent following Sputnik's launch in 1957 on developing systems to automatically translate Russian documents into English. This turned out to be a very hard problem, since much specialized knowledge seems to be required to understand language. A famous example:

> *The spirit is willing but the flesh is weak.*

was translated into Russian and then retranslated back into English, giving:

The vodka is strong but the meat is rotten.

At that point, people realized two things that made the AI problem much harder than they had originally thought.

(1) In order to do a good job in any realistic task, simple syntactic manipulation (i.e., simple rules to shuffle words around or do Russian-to-English dictionary lookups) is not good enough. Instead, we must have enough knowledge about the world to really understand what's being said, so as to reason more deeply about it. For example, in the translation example, we need to understand that 'spirit' refers to the metaphorical or mystical human spirit, rather than to alcohol.

(2) Computational intractability. The AI goal was defined before the theory of NP-completeness was developed. At that point, people thought that to deal with larger problems, we need only larger/faster computers. In particular, the phenomena of exponential scaling – in which the computation scales exponentially with the size of the problem – was not yet understood. Many early AI methods required solving NP-hard problems, and therefore did not scale well to larger problems.

1.2 Approaches to AI?

How about the state of AI today? As a field, AI is now significantly more mature, and we have a better understanding of what sort of methods work and might scale well. There are perhaps two broad approaches to developing AI methods today: (i) Acting like humans, and (ii) Acting rationally.

1.2.1 Acting like humans

The Turing Test, illustrated in Fig. 1.1, is perhaps the most famous example of the former: a machine is pronounced intelligent if it is indistinguishable from a human. Today, very little serious work is actually directed specifically at passing the Turing test.

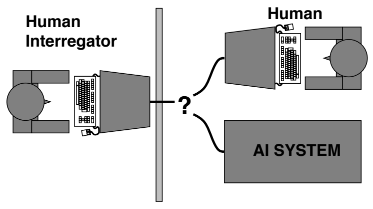


Fig. 1.1 Illustration of Turing's imitation game.

Alan Turing anticipated many major arguments against AI in the following 50 years and predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes. He also suggested major components of AI: knowledge, reasoning, language understanding, learning, which are now the backbone of modern AI.

There is however a small (but growing) community of researchers that hope to achieve human-level AI by using insights from the humans – specifically, from the human brain. This line of research is inspired by the thesis that much of the human brain may be implementing a learning algorithm. Inspired by this thesis, several research groups are trying to elucidate what the brains learning algorithm might be, and implement this algorithm (or an approximation to it) on a computer, so as to perhaps take a baby step towards solving the problem of building machines that have intelligence comparable to humans.

Understanding how the brain and cognition works (computational neuroscience: *how the brain thinks*) is an important area in its own right (and subject of CSCI 6508). After some years of clear separations between AI and computational neuroscience, there is now a renewed convergence with exciting new ideas.

1.2.2 Acting rationally

Rather than trying to get computers to act like humans, the majority of AI researchers have instead focused on the second approach, of trying to get computers to act rationally. This class of methods will be the focus of this course.

Rational behavior means hereby *doing the right thing*. The right thing is thereby a set of actions that is expected to maximize goal achievement, given the available information. This does not necessarily involve thinking, but thinking should be in the service of rational action.

Today, many AI systems consider *rational agents*, where an *agent* is generally an entity that perceives and acts. In an abstract form, an agent often be described as a function from percept histories to actions:

$$[f : \mathcal{P}^* \rightarrow \mathcal{A}]$$

For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance. Importantly for practical considerations, there are typically *computational limitations* which make perfect rationality unachievable. Thus, in practice we are mostly interested in designing the best *program* for the given se of machine resources.

The paradigm of acting rationally has seen numerous successes in terms of building very useful AI systems with significant societal and economic impact. In fact, you probably use AI algorithms dozens of times a day without being aware of it, such through using web search engines, sending US mail or writing checks (where software reads zip codes or handwritten checks automatically), finding driving directions online, receiving Amazon/Netflix/etc. recommendations for books or movies you might like, fraud detection algorithms that check whether your credit card purchases are legitimate, spam filters, and many more. At a high level, one can view this type of AI as being composed of a set of techniques, such as search, machine learning, constraint satisfaction, and probabilistic models. These techniques are useful for a variety of tasks that are necessary to building various intelligent systems, such as the problems of perception (understanding the physical environment using its sensor inputs), planning, navigation, etc. This is a many-to-many relation: Many very different techniques can be used to perform the same task, and one technique is useful for a wide variety of tasks, and as a component in other techniques.

1.3 AI areas

- State graphs & search
  - Uninformed search
  - Heuristic search
- Knowledge representation & expert systems

  - Formal logic (propositional, first-order)
  - semantic nets
  - case-based reasoning

- Machine learning & probabilistic reasoning
  - Artificial Neural Networks & Support Vector Machines
  - Bayesian networks
  - Hidden Markov models & Kalman filters
- Common concepts & applications
  - Intelligent (rational) agent systems
  - Planing and decision making
  - Natural language processing
  - Games

In this course we will start with formulating AI applications as search problems, and talk about basic search strategies. We will then focus on machine learning where much recent progress has been made. This will include supervised and unsupervised learning, some introduction to probabilistic reasoning, and some further outlook to recent research in this area.

Part I

Search

Robotics and motion planning

2

To build the controller for a rational agent (robot), we need to find a way to describe the problem in a suitable way. In other word, we need to learn how to translate a real world problem into a description that we can handle with a computer. Central for this task is the understanding of *abstractions* and configuration space.

An intelligent systems can be a robot and have a physical presence, or it can live within your desktop computer. Today, let's talk about what it takes to physically control a robot. What is a robot? Informally, it is a physical system that interacts with the environment using physical sensors and effectors. Some examples of robotic sensors include video cameras; sonar (which measures distances to obstacles by measuring how long it takes sound to bounce off the obstacle and return to the robot); laser range scanners (which works similarly to sonar, except it bounces light rather than sound off the obstacles); microphones; odometers (which measure distance traveled); GPS; accelerometers; and many more. These sensors give the robot information about the state of the environment around it, as well as information about the robots location and orientation within that environment. Another important type of sensor are proprioceptive sensors, which tells the robot about the position or changes in position of its own joints. For most robots, their effectors can be divided into two categories based on their function:

- Locomotors, such as wheels or legs, to allow the robot to move itself around. Wheels are popular since they are easier to control, but there many other possibilities, such as legs.
- Manipulators, such as a robot arm and hand, which allow the robot to interact with the world and affect the world around it.

Given a robot, how can we get it to drive from one place to another, without hitting obstacles? Alternatively, given a robot arm, how can we generate a smooth motion for it to reach out and, say, pick up an object? At its basic level, a robot consists of a set of motors, and we have to decide (say) what sequence of joint angles to command each motor to go to. We would like to find a sequence of joint angles that will cause the robot to follow some path from its initial position to some goal position. To develop algorithms to accomplish this, we will need to introduce the concept of a configuration space.

2.1 Configuration or state space

First consider a robot in a 2D plane. How would we describe its position? That depends on whether it can rotate, or just translate without rotating in the plane. In the latter case, we can describe the robots position with a pair of real numbers, such as its Cartesian coordinates (x, y). In the former case, we would need three real-valued parameters (x, y, θ), with θ giving the robots orientation.

This leads us to the important notion of *degrees of freedom (dof)*: A robot has k degrees of freedom if its current *configuration* can be fully described by a set of k real numbers. Thus, the robot which is allowed to translate and rotate has three degrees of freedom, while the one which is only allowed to translate in a plane has two.

How about the state of a helicopter, that can fly around in 3D? We would need three real numbers to give its position in space, as well as three more numbers (such as roll, pitch, and yaw) to specify its orientation. Thus, the helicopter has six degrees of freedom. Note that there are sometimes many possible choices for the parametrization; for instance, we could use either the Cartesian coordinates (x, y, z), or latitude, longitude, and height above sea level to specify the location. The relative merits depend on the application, but any sensible parametrization should still result in six degrees of freedom.

What about the robot arm shown in Fig. 2.1a? One (naive) parametrization would be to give the Cartesian coordinates of the lower-left corners of each of the two parts; thus we could specify the position of the robot using four real numbers. The problem with this parametrization is that that almost all values of these four real numbers correspond to illegal configurations for the robot (and are not possible unless we break the arm); therefore, it doesnt capture the true dimensionality of the space. A better parametrization would be two real numbers specifying the angle of each of the joints. This captures the allowable variation with the minimum number of dimensions.

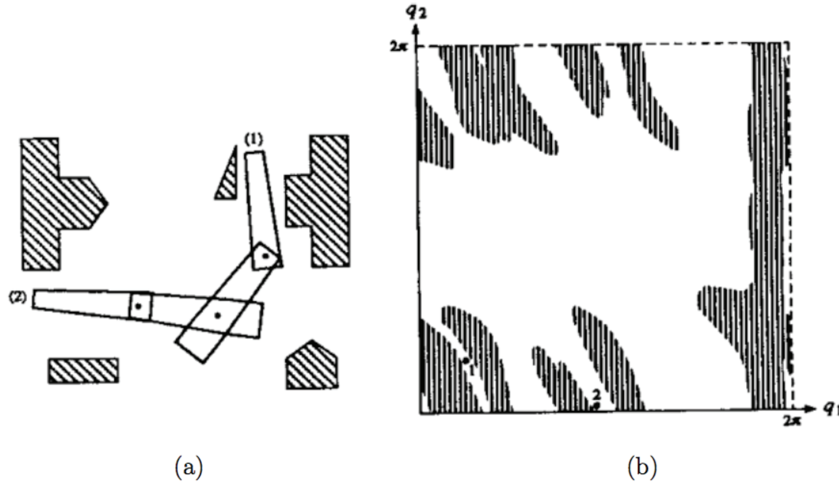


Fig. 2.1 (a) The workspace for a robot arm with two degrees of freedom, one for each joint. (Assume the base of the arm is anchored to the wall.) (b) The configuration space for the robot. Images courtesy Jean-Claude Latombe.

Suppose a robots configuration can be specified via k real numbers. The set of all possible values for these k real numbers is called the *configuration space* or *state space* of the robot. Consider, for instance, a circular robot which moves in a 2D workspace as shown in Fig.2.2a. The configuration space for this robot is the set of all (x, y) coordinates; thus its configuration space is $\mathbb{R}^2$.

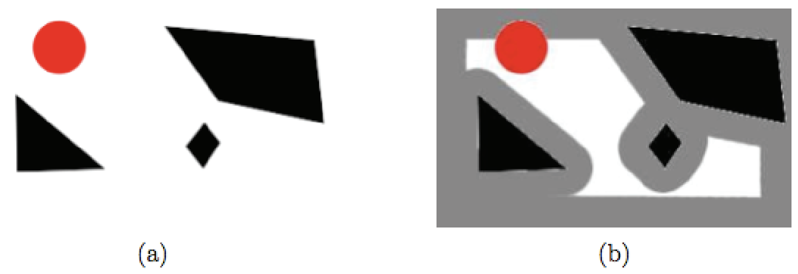


Fig. 2.2 (a) An example workspace with the circular robot shown in red. (b) The resulting configuration space, where the white areas are the free space. Coordinates are defined with respect to the center of the robot.

Our goal is path-planning – finding a legal path for the robot which it can physically follow. Therefore, we will divide the configuration space into two parts: *free space* and *obstacles*. Free space is the part of the configuration space that represents legal positions of the robot; obstacles correspond to the illegal positions, such as if the robot is physically embedded within some other object.[1]

[1]Note that obstacles in the configuration space may not necessarily correspond only to physical obstacles in the world; for example, they might also correspond to states where one part of the robot tries to pass through another part.

In point of terminology, we will use the term *workspace* to refer to the physical environment that the robot works in. Thus the workspace is always 2D or 3D, and corresponds directly to our physical world. The term free space will always refer to a subset of the configuration space. But we may use obstacle either to refer to the set of illegal positions in configuration space, or to physical obstacles that the robot may encounter in its workspace. Previously, we described a 2D planar robot. What is its free space? That depends on how big the robot is. If the robot is a point, the configuration space looks the same as the workspace; but if the robot has shape, the two usually differ. In this case, we need to be precise about how we use a pair of cartesian coordinates (x, y) to describe where the robot is. We define the configuration of the robot by the location of some specific reference point on the robot. If the robot is round, for instance, we may choose its center as the reference point.

For the case of a non-rotating 2D robot, a point (x, y) in the configuration space is in the free space if, when the physical robots reference point is at position (x, y) in the workspace, the robot does not intersect with any obstacle in the workspace. The free space for our robot is shown in Fig. 2.2b. What about our arm robot from before? There, we have to be even more careful. Obstacles in the configuration space include configurations where the robot overlaps an obstacle, but also configurations where one arm of the robot passes through the other. The resulting configuration space is shown in Fig. 2.2b.

2.2 Path planning

Recall that our task is to find a path moving the robot from an initial position to some goal position. The initial and goal positions of robot will correspond to a pair of initial and goal points in the configuration space. To accomplish our task, all we need to do therefore is find a path in configuration space going from the initial configuration to the goal configuration, that is contained entirely in the free space. Once we have found a path, the sequence of points along this path will correspond to a continuous sequence of valid positions for the robot that take it from the initial position to the goal position. This formulation of the problem makes several assumptions, namely, that:

- The obstacles are known in advance (and do not move).
- The robot can follow any path in free space.

These assumptions often hold, but lets discuss the second of these as- sumptions in more detail. In particular, lets discuss an example of when the second assumption is violated.

A car moving in a 2D plane has three degrees of freedom, since its state can be represented with the triple (x, y, θ), where θ is its orientation. We really do need all three degrees of freedom to represent the cars position, since given a big enough plane, it can get itself into any position and orientation. Suppose now that we have a car in position (x, y, θ), and we want it to wind up in $(x, y + \Delta y, \theta)$. Assuming there are no obstacles in the workspace, there is clearly a straight line in configuration space connecting these two points: one in which y varies, but x and θ remain constant. The car can follow the path if $\theta = 90^o$ or 180^o, because the car is then aligned with the y axis. Otherwise it is impossible, because that would require it to move sideways.

In this particular case, we have three degrees of freedom, but we cant control them directly. In fact, we only have two independent controls: forward-backward motion and steering. We say that the car has only two *control-lable degrees of freedom.* Thus, even given two adjacent points in configuration space, we may not be able to move directly from one of these points to another. A robot where the number of controllable degrees of freedom is equal to the number of total degrees of freedom is called *holonomic.* Otherwise, its called *non-holonomic.* It is possible, albeit difficult, to design robots capable of holonomic locomotion in a plane. There are also techniques for controlling nonholonomic robots (some of which well see later this quarter). But for simplicity, we will assume here that our robot is holonomic.

2.3 Search space for motion planning

We have seen that the motion planning problem, under certain assumptions, can be reduced to the problem of finding a path from an initial configuration to a goal in the robots configuration space. Note how elegant this formulation is – whereas the problem of moving a robot arm from one position to another seems to involve lots of complicated things such as the geometry of the arm, what obstacles there are around the robot, the specific shapes/positions of these obstacles, etc., the notion of a configuration space takes all of the geometry and obstacles into account, and reduces the motion planning problem into that of planning a path in a k-dimensional configuration space.

Unfortunately, the configuration space is continuous, and it is usually very hard to reason directly with continuous spaces. Therefore, our approach to finding a path in configuration space will be based on formulating a discrete graph search problem, and applying standard discrete graph search algorithms to find a path. The problem of finding a path (or the shortest path) from an initial state to a goal in a discrete graph is well-studied in computer science.

There are many ways to discretize a continuous configuration space, and the relative merits of each depend on many factors of the particular task, such as the number of degrees of freedom, the complexity of the obstacles, and the computational resources available. We are particularly interested in three properties of a discretization method and/or search algorithm:

(1) **Completeness:** if a path exists in continuous space, whether our algorithm will find a path.
(2) **Optimality:** if a path exists, whether our algorithm will find the shortest one.
(3) **Computational complexity:** how the running time grows as a function of the problem size. Many algorithms work fine for small numbers of dimensions (such as 3 or 4), but dont scale well to higher dimensions.

Our approach will be to apply a *roadmap* methods, also called *skeletonization* methods. Specifically, we will create a discrete graph where vertices in the graph correspond to points, called *landmarks*, in the configuration space. The edges in the graph will correspond to paths (such as straight lines in configuration space) between the landmarks.

2.3.1 Grid discretization

To apply a grid discretization, we choose the landmarks to be the set of points lying in a regular grid in the configuration space (discarding points that lie in obstacles rather than free space). One vertex of our discrete graph will correspond to each of these landmarks. In addition, two adjacent vertices in the grid are connected if there is a straight-line path between them that lies entirely in free space.

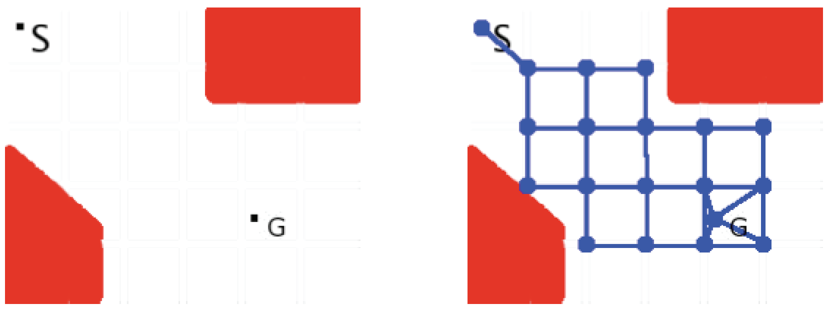


Fig. 2.3 (left) An example of using a grid to select landmarks in a configuration space. The obstacles are shown in red. (right) The resulting graph is superimposed.

For any fixed resolution grid, this method is unfortunately not complete or optimal. [2] Grid based discretization pays a hefty price, however, in terms of

[2]By modifying the algorithm to keep on trying finer and finer grids, it is possible to get

computational complexity. For instance, if we have a 3D configuration space, and discretize each axis using 512 values, we get $512^3 \approx 1.3 \times 10^8$ vertices. For 10 dimensions, if we discretize each dimension with just 100 values, we get on the order of 10^{20} grid cells.

Grid based discretization often works fine for low dimensional problems (say 2D to 4D configuration spaces). For slightly higher dimensional problems (say 5D-6D) you can sometimes get it to work if youre clever and choose the grid very carefully. But for problems much higher dimensional than that, grid discretization usually does not work well, because of the exponential blowup in the number of dimensions.

2.3.2 Visibility graph

Particularly in high dimensional configuration spaces, using a regular grid results in choosing far two many landmarks. This and the following section describe two ways for choosing landmarks that result in a significantly smaller number of them.

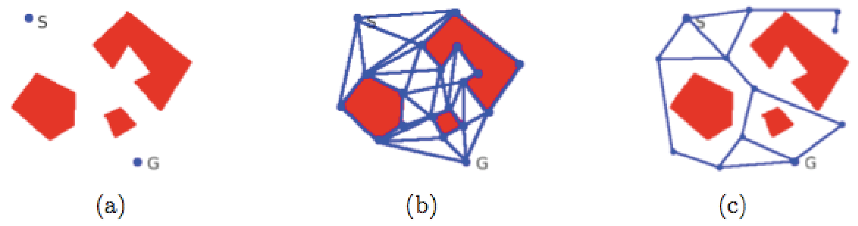


Fig. 2.4 (a) A path planning problem, where we need to get from initial state S to goal state G. (b) The visibility graph for this planning problem. (c) An example of a probabilistic roadmap. For clarity, not all edges are shown.

In the *visibility graph* method we assume that all the obstacles in the configuration space are polygons.[3] We choose as landmarks all of the vertices of all of the obstacles in the configuration space (plus the start and goal states).[4] We then draw edges between all pairs of vertices such that one is visible from the other, i.e., that can be connected by a straight line. An example is shown in Figure 4 (b). It is possible to prove that the visibility graph method is complete.

In general, most obstacles are not, however, polygons (for example, see Figure 1b). So the visibility graph method is not very widely applicable.

2.3.3 Probabilistic Roadmap planning

The Probabilistic Roadmap (PRM) algorithm is due to Jean-Claude Latombe, and is widely used for many classes of robots, such as many robot arms. It is a randomized planning algorithm, in which we choose points at random in the

[3] If you really want to get the terminology right, polygons are 2D; their generalization to 3D is called polyhedrals; and their generalization to arbitrary numbers of dimensions is called polytopes.

[4] One problem with this approach is that the resulting path tends to follows the edges of the obstacles; this corresponds to having the robot just barely brush against the edges of the obstacles, and is undesirable, since robots cant be controlled to perfect accuracy. In practice, for robustness, the obstacles are usually expanded slightly when the graph is generated.

the algorithm to be complete – since if we sample finely enough, we will get good enough resolution to find a path if one exists. By modifying the algorithm even further (to connect all pairs of cells for which the straight-line path between them is entirely in free space, rather than only immediately adjacent landmarks), it is also possible to get the algorithm to become arbitrarily close to optimal, since as we use finer and finer discretizations, there will exist a path arbitrarily close to the optimal one.

free space to be landmarks. Concretely, we sample points at random from the configuration space, and then discard the ones which lie within obstacles; this leaves us a set of points lying in free space, we will be the landmarks. We then check each pair of (say) nearby landmarks to see if they can be connected by a straight line that lies entirely in free space. An example is shown in 4 (c).

In the PRM algorithm, we cant guarantee completeness or optimality, because we could, in principle, get extremely unlucky and choose a very bad set of landmarks. However, it is possible to make probabilistic guarantees on completeness. For instance, assuming that the obstacles are widely spaced apart, it is possible to show that a path from the initial state to the goal will be found with very high probability (assuming one exists) so long as the number of samples is sufficiently large.

Finally, note that in all of these algorithms, we dont ever explicitly compute what the entire configuration space is or otherwise come up with a complete explicit representation of what are the free space and obstacles in configuration space. Thus, even though weve been drawing examples of 2D configuration spaces in these lecture notes and on the chalkboard during lectures in order to illustrate these ideas, you wouldnt actually write a program to draw out an entire k-dimensional configuration space (which is not re- alistic to do anyway when k is large, because of the exponential scaling in k). Specifically, note that in order to implement PRMs, all we need are (i) A way to sample points randomly in configuration space, (ii) A way to test if each of these points lies in free space or obstacle, and (iii) A way to test if the straight line between a pair of these points also lies entirely within free space. Step (i) is easily done using a random number generator, and steps (ii) and (iii) are standard geometric calculations (for example, in the case of a robot arm, step (ii) would typically require only checking whether the robot will intersect either with itself or with any obstacles in its workspace when all its joint angles are set to specific values); and therere many free software packages that you can use to perform these purely geometric computations.

2.4 Abstraction

In the examples above it was described how to replace a continuous configuration space with a discrete one so that standard search algorithms can be applied to the robot problem. In general, the real world is absurdly complex and the configuration space (or state space) must be *abstracted* for problem solving. Also, real world actions have to be abstracted so that they can be represented by a computer. For example, the action of traveling from Halifax to Hawaii is a complex thing, including not only different transportation means, emotional states of travelers, luggage, logistic, etc. However, depending on the task to be solved, not all such details are ecessary.

Search

3

This chapter is on search and is divided into three parts

(1) Uninformed search (tree search, graph search, etc)
(2) Heuristic search (A*, etc)
(3) Optimization search algorithms (gradient decent, GA, etc)

3.1 CS221 Lecture notes by Andrew Ng, No. 3

CS221 Lecture notes #3

Search

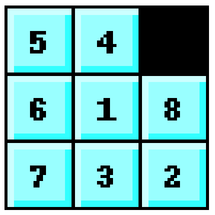
Previously, we showed how discretization techniques, such as grids, visibility graphs, and probabilistic roadmaps, could be used to convert a continuous motion planning problem into one on a discrete graph. How can we search graphs like these efficiently? In this set of notes, we will present algorithms to solve search problems on discrete graphs. We will first discuss **blind search** (also called **uninformed search**) algorithm, where we know nothing except for the nodes and edges which make up the graph. We will then describe **heuristic search**, in which we will use knowledge about the problem to greatly speed up the search. These search algorithms are quite general, and are widely used many areas of AI.

Throughout much of these notes, our motivating example will be a toy problem known as the 8-puzzle, shown in Figure 1.

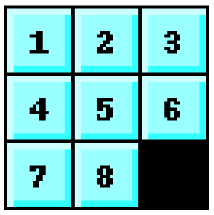
1 Search formalism

A discrete graph search problem comprises:

- **States**. These correspond to the possible states of the world, such as points in configuration space, or board positions for the 8-puzzle. We typically denote individual states as s, and the set of all states as S.
- **(Directed) edges**. There is a directed edge from state s_1 to state s_2 if s_2 can be reached from s_1 in one step. We assume directed edges for generality, but an undirected edge can be represented as a pair of directed edges. We typically use e to denote an edge, and E the set of all edges.
- **Cost function**. A non-negative function $g : E \mapsto \mathbb{R}_0^+$. (This notation means g is a function mapping from the edges E into the set of non-negative real numbers $\mathbb{R}_0^+$.)



(a)



(b)

Figure 1: The 8-puzzle. There are 8 tiles, numbered 1 through 8, which slide around on the board. In the initial state, the tiles are scrambled. The goal is to put the numbers in order, with the blank square in the lower right hand corner. (a) An initial state. (b) The goal state.

- **Initial state**. Usually a single state $s \in S$.
- **Goal**. For generality, we represent the goal with a **goal test**, which is a function that tells us if any particular state s is a goal state. This is because our task may be to get to any state within some "goal region," rather than to a very specific, single, goal state. For instance, in motion planning, the goal test might be "end of finger presses the elevator button." This cannot be described with a single goal state, since many different configurations of the robot's joints are probably consistent with the goal. In problems where we are interested in reaching one particular goal state, the goal test will be a function which returns true for a state s if and only if it is the goal state.

Given this definition of a search problem, there are two ways that we can represent the search space:

- **Explicitly**. In this case, we explicitly construct the entire graph of the search space in computer memory or on disk. Thus, we would create a list of all the states, all the edges, and all the costs. The goal test will also be explicitly represented as a list of all the goal states. This explicit representation only works for fairly small graphs. Consider, for instance, our example of the 8-puzzle. The total number of possible board configurations is $9! = 1 \times 2 \times \cdots \times 9$, and the total number of edges is larger still. For the larger 15-puzzle (on a 4x4 grid rather than 3x3 grid), the number of states is $16! \approx 2 \times 10^{13}$. Clearly, we can store the graph in memory only for the smallest problem sizes.
- **Implicitly**. To represent a search problem implicitly, we will use some data structure for representing individual states. For example, if the

states comprise a 2D grid, the data structure might be an (i, j) pair. The edges and costs are also represented implicitly. Edges are usually described in terms of **operators**, which are functions $o : S \mapsto S$ that map from states to states.[1] In a 2D grid, our operators may be N, S, E, and W, corresponding to movements in the four directions. In the 8-puzzle, we would have some data structure that represents board positions, as well as four operators, each corresponding to moving one of the four adjacent tiles into the empty hole. Each operator is assigned a positive real-valued cost.[2] The **successor states** of a state are the states reachable by the application of a single operator.

2 Basic search

As we develop different search algorithms, we'll be interested in questions of whether they are complete, whether they are optimal, and in their computational efficiency. A search algorithm is **complete** if, whenever there exists a path from the initial state to the goal, it will find one. It is **optimal** if any path it finds to the goal is a minimum cost path.

We first consider the case where we are given nothing except the nodes, edges, and costs of the search space. This is referred to as **blind**, or **uninformed** search. We will discuss several different search algorithms for discrete graphs (represented either implicitly or explicitly). Our search algorithms will, conveniently, all follow the same basic search template:

```
Queue q;
q.insert(initialState);
while (!q.isEmpty()){
  node = q.remove();
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators))
    q.insert(n);
```

[1]We will be slightly loose in our terminology, and not specify the result of operators which cannot be applied in a given position, such as what happens if the N operator is applied when we're at the uppermost row of the grid and can't move any further north. There are many options: the operator could be deemed inapplicable to that state, it could return the same state, and so on.

[2]We could allow the cost also to depend on the state itself. For instance, it might cost a robot dog more to move north in rugged terrain than smooth terrain. This is a reasonably straightforward generalization that would require only a small change to the algorithms we describe below.

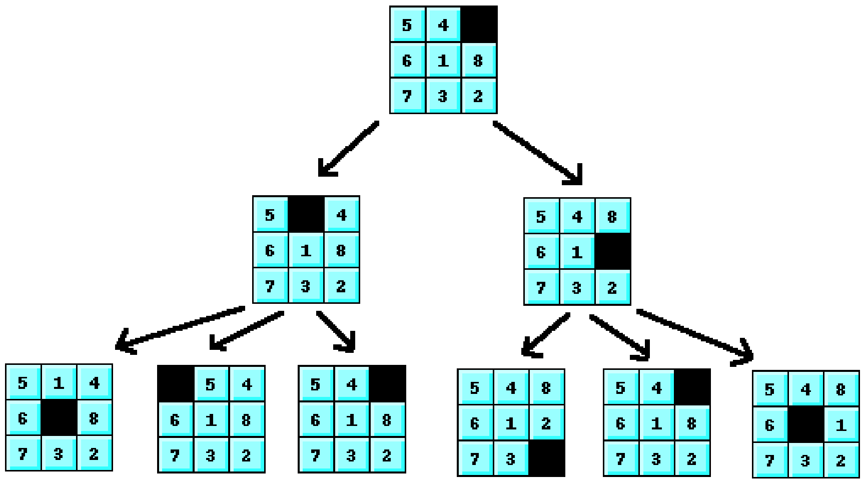


Figure 2: A portion of a search tree for the 8-puzzle. The root node is the initial state, and each of the children of a given node corresponds to one of the four possible operators. In this example, we did not eliminate repeated states (which you should usually do); so, for example, the third and fifth nodes in the bottommost row are repeats of the root.

```
}
return FAIL;
```

For now, we will leave exactly what sort of queue we use unspecified. Depending on this choice, we will get different search algorithms.

In the inner loop of the algorithm, we remove a state, and check if it's a goal. If it's not, we put all of its successors on the queue, to be examined later. We say a node is **expanded** when it is taken out of the list and checked for goalness and its successors are inserted into the queue.

Note that the order of operations is important. In particular, we check for goalness when the node is expanded, *not* when it is being added to the queue. This will become important later on when we prove optimality of various algorithms.

This process implicitly defines a search tree, like the one shown in Figure 2. The search tree contains **nodes** n. Each node is labeled with a state s. Each node also corresponds to a specific *sequence of states* (specifically, the sequence of states you pass through as you start from the root, and traverse the tree until you reach n). Because this is a tree, the path from the root to a node n is always unique. Note that even though nodes in the tree are labeled with states (board positions), nodes and states are two distinct concepts.

We will use "states" only to refer to states in the original search graph, and "nodes" only to refer to nodes in the search tree. For example, in the original search graph, there can sometimes be multiple paths from the initial state to some other state s. Thus, as shown in the figure, the search tree can also contain multiple nodes n that are labeled with the same state s. (Later in these notes, we'll also discuss how to deal with repeated states, but we'll ignore this issue for now.)

The search tree is defined as follows. When the search algorithm is first run and the initial state is placed on the queue, think of this as constructing a tree with just a root node, which is labeled with the initial state. At each point in our search, the nodes in the queue correspond to the "fringe" of the search tree, and nodes which have already been expanded correspond to the interior nodes. Each time we expand a node on the queue, that corresponds to taking a node on the fringe and adding children to it, and then putting these new children (which are now on the fringe) onto the queue.

Note that this search tree is a mathematical concept used to help us reason about the process of running the search algorithms, and is implicitly defined by the running of our search algorithm. The search tree isn't actually explicitly constructed anywhere.

Now we discuss particular search algorithms which follow this template. We give only a very brief overview of each algorithm; some additional details can be found in Section 3.4 of the course textbook. Also, see the Exercise Set 1 solutions for worked-out examples of each.

2.1 Depth-first search

When our queue is a LIFO queue (stack), we get **depth-first search (DFS)**. DFS keeps recursively expanding nodes until it reaches a goal state or encounters a state with no successors. When a state has no successors, DFS backtracks.

In an undergraduate programming/data structures classes, you may have seen DFS written as follows, where the stack was implicitly maintained through the program's stack frame:

```
State DepthFirstSearch(node){
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators)){
    result = DepthFirstSearch(n);
    if (result != FAIL) return result;
  }
  return FAIL;
}
```

Using a LIFO queue in the search template given previously, the stack is now explicitly maintained through our own queue, but it results in the same search process as this.

Under mild assumptions, DFS is usually complete. Unfortunately, DFS is generally not optimal.

2.2 Breadth-first search

When our queue is a FIFO queue, we get **breadth-first search (BFS)**. Intuitively, nodes will be expanded in order by the length of the path (number of edges). (You should convince yourself of this.)

BFS is a complete search algorithm under very reasonable assumptions. It is also optimal in the particular case where all of the operators have a cost of 1 (because it then becomes a special case of uniform-cost search).

2.3 Uniform-cost search

When our queue is a priority queue ordered by the total cost of the path to a node, we get **uniform-cost search (UCS)**. Note that by the definition of a search tree, the path from the root node to any given node n is always unique (even if, in the original search graph, there may be multiple paths from the initial state to the state s associated with the node n). Thus, the priority of n is simply the total cost of all the edges between the root node and the node n. We will denote this cost $g(n)$.

Assuming that we handle repeated states (described in Section 4.1, this algorithm is also commonly known as Dijkstra's shortest-path algorithm, which is a complete and optimal algorithm. (You can find a proof of this in the textbook.) The gist of the optimality proof is as follows: nodes are expanded in order according to the total cost to reach that node. Out of all paths to a goal, the optimal path is (by definition) the shortest, and so the node in the search tree corresponding to that path will be expanded before all of the others. This proof works for all finite graphs, as well as infinite graphs under mild assumptions.

3 Heuristic search

All of the algorithms presented above are examples of blind (uninformed) search, where the algorithm has no concept of the "right direction" towards the goal, so that it blindly stumbles around until it happens to expand the goal node. We can do much better if we have some notion of which is the right direction to go. For instance, if a robot is trying to get from the first floor to the basement, then going up in the elevator is unlikely to help. One way to provide this sort of information to the algorithm is by using a **heuristic function**, which estimates the cost to get from any state to a goal. For instance, if our goal is to get to the basement, a heuristic might assign a higher estimated cost to the fourth floor than to the first. More formally, a heuristic is a non-negative function $h : S \mapsto \mathbb{R}_0^+$, that assigns a cost estimate to each state.

For example, consider the grid in Figure 3. We know it is likely to be more productive to move right than to move left, because the goal is to the right. We can formalize this notion with a heuristic called **Manhattan distance**, defined as the total number of N/S/E/W moves required to get from the current state to the goal, *ignoring obstacles.* The name "Manhattan distance" comes from the observation the in much of Manhattan island in New York, the city blocks are our laid along the compass directions, out so that you can only drive in compass directions, rather than diagonally.

In constructing heuristics, we often face a tradeoff between the accuracy of a heuristic and how expensive it is to compute it. For instance, here are two examples of trivial heuristics:

- The constant heuristic $h(s) = 0$. This heuristic requires no computation, but provides no useful information about the problem.
- The heuristic equal to the minimum cost from s to a goal. If we knew this heuristic function, the search problem would be trivial. (Why?) However, computing this requires actually solving the problem, and so it is no good in practice.

The most useful heuristics will almost always lie between these two extremes.

So far, we have defined a heuristic function as taking input a state. Given a node n in a search tree, we also define $h(n)$ to be the value obtained by applying the heuristic function h to the state s that the node n is associated (labeled) with.

Let us now consider some particular algorithms which make use of the heuristic function. As in the section on uninformed search, we will follow the search template given previously.

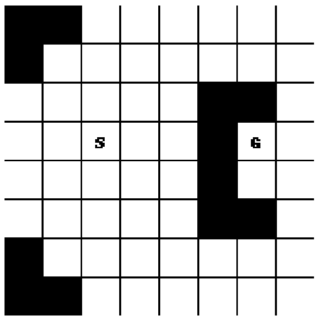


Figure 3: A search problem on a grid, with obstacles in black.

3.1 Best-first search

As in uniform-cost search, we will take our queue to be a priority queue, but this time the priority is given by h itself. The resulting algorithm, **best-first search** (or **greedy search**), greedily expands the node n with the smallest value of $h(n)$, i.e., the one with the smallest estimated distance to the goal. Thus, it repeatedly picks whichever node on the fringe is estimated to be closest to the goal, and expands that node.

Unfortunately, best-first search is not optimal, as is demonstrated in Figure 4. Often, the node which is seemingly closest to the goal is already the result of a long search path. For example, let's suppose we have a choice between expanding two nodes m and n, where the path to get to m is of length 8, and the heuristic value is 3, while the path to n is of length 19 and the heuristic value is 2. Best-first search would choose to expand n, because it looks closer to the goal. However, any resulting path must be of length at least 21, while expanding m might lead to a path of length 11. The problem with best-first search is that it only considers the expected distance to the goal, and this causes it not to be optimal.

By modifying the algorithm to also take into account the cost already incurred along a path, we can obtain an algorithm that is optimal (in the sense of finding minimum cost paths to the goal).

3.2 A^* search

We now describe the A^* **search** algorithm. This algorithm is due to Nils Nilsson, and is one of the most widely used algorithms in all of AI. It uses uses a priority queue like uniform cost search and best-first search, but the priority of a node n is now given by

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the total cost of the path from the root to node n, and $h(n)$ is the heuristic as before.

In other words, A^*'s priority value for a node will be our total estimated cost of getting from the initial state to n to the goal. Thus, the priority takes into account two costs:

- The cost to get to the node n, given by $g(n)$.
- The cost to get from n to a goal, which is estimated by $h(n)$.

Uniform cost search took into account only the first term, and being a blind search algorithm, can be very slow. Best-first search took into account only the second terms, and was not optimal.

For the sake of completeness, here is pseudocode for the A^* algorithm:

```
PriorityQueue q;
q.insert(initialState, h(initialState));
while (!q.isEmpty()){
  node = q.remove();
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators))
    q.insert(n, g(n) + h(n));
}
return FAIL;
```

Let us now prove some key properties of A^* search For convenience, we have summarized most of the notation we'll use in this set of notes in Figure 5.

4 Optimality of A^* search

One basic question to ask about any search algorithm is whether it is **complete**. Recall that a search algorithm is complete if it is guaranteed to find some path to a goal state whenever such a path exists. Under fairly mild assumptions, we can show that A^* is complete. We will not present the proof here, but the interested reader is referred to Chapter 4 of the textbook.

A more interesting question is **optimality**. A search algorithm is optimal if it is guaranteed to find the least-cost path to a goal state, provided a path to the goal exists. To prove the optimality of A^* search, we first need a definition.

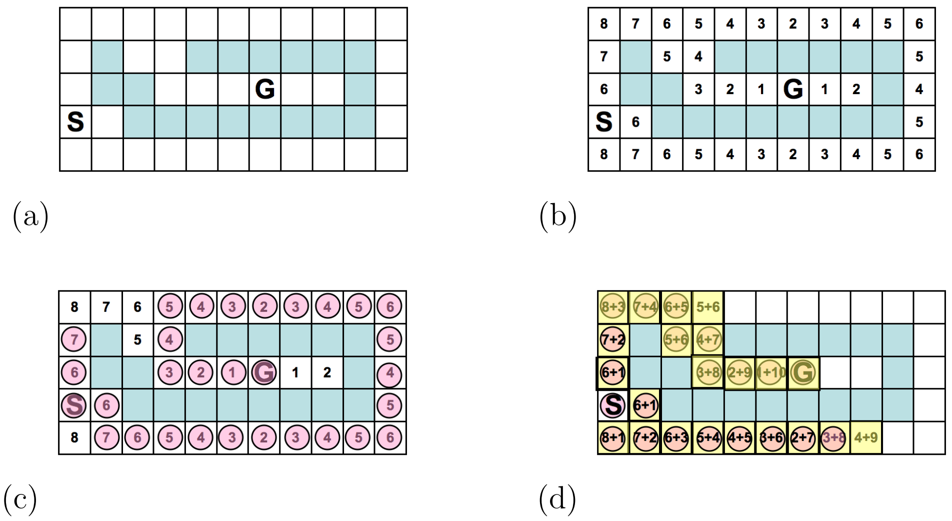


Figure 4: (a) An example of a search problem where the goal is to get from S to G. (b) The search space, labeled with the Manhattan distance heuristic. (c) The set of nodes expanded by best-first search. The solution is clearly suboptimal. (d) The set of nodes expanded by A^* search. (Note: the pair of numbers in each square represent the heuristic value $h(n)$ and the distance already traveled $g(n)$.)

s	a state in search space
n	a node in the search tree
n_g	A goal node
$g(n)$	The cost of the path to node n
$h(n)$	The heuristic function evaluated at n
$h^*(n)$	The actual cost of the least-cost path from n to a goal state
$f(n)$	The estimated cost of the least-cost path that goes from the root node through n to a goal, $f(n) = g(n) + h(n)$
$f^*(n)$	The actual cost of the least-cost path that goes from the root node through n to a goal, $f^*(n) = g(n) + h^*(n)$
$Pa(n)$	The parent of node n in the search tree.

Figure 5: Notation used in this lecture.

Definition. A heuristic function h is **admissible** if it never overestimates the true cost to get to the goal; in order words, if for any state s, we have $h(s) \leq h^*(s)$.

Theorem 4.1: *If h is an admissible heuristic function, then A^* search with h is optimal.*

Proof: Our overall strategy is to take n_g, the first goal node to be expanded, and show that it represents an optimal path. Since A^* returns the first goal node expanded, this implies the optimality of A^*.

Suppose h is an admissible heuristic, and n_g is the *first* goal node to be expanded. Since n_g is a goal and h is an admissible heuristic, we have

$$\begin{aligned} 0 &\leq h(n_g) && \text{since heuristic functions are nonnegative} \\ &\leq h^*(n_g) && \text{since } h \text{ is admissible} \\ &= 0 && \text{since } n_g \text{ is a goal} \end{aligned}$$

Hence, $h(n_g) = 0$, and therefore

$$f(n_g) = g(n_g) + h(n_g) = g(n_g). \tag{1}$$

We need to guarantee that the path to n_g is no longer than any potential path through some other unexpanded node. Suppose that (by expanding more nodes) it is possible to reach some other goal node n'_g other than the one n_g chosen by A^*. We will prove that $g(n'_g) \geq g(n_g)$.

There is some unique path from the root to n'_g. Let n be the *first* node on this path that has not yet been expanded. Note therefore that n's parent must have been expanded previously, and therefore n is on the fringe of our search tree at the instant that A^* expanded n_g.

Let $f^*(n)$ denote the minimum cost path to a goal that passes through n. Since n'_g is a goal node, and the path to n'_g passes through n, we must have

$$f^*(n) \leq g(n'_g).$$

So, by admissibility, we have

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f^*(n) \leq g(n'_g). \tag{2}$$

Furthermore, n is on the fringe, and was therefore on the priority queue (along with n_g) at the instant A^* expanded the goal node. But A^* chose to expand n_g rather than n. Since A^* chooses nodes to expand in order of priority, this implies that

$$f(n_g) \leq f(n). \tag{3}$$

Putting together equations (1-3), we get

$$\begin{aligned} g(n_g) &= f(n_g) && \text{by Equation (1)} \\ &\leq f(n) && \text{by Equation (3)} \\ &\leq g(n'_g) && \text{by Equation (2)} \end{aligned}$$

This proves that $g(n'_g) \geq g(n_g)$ for any goal node n'_g other than the one chosen by A^*. This therefore shows that A^* finds a minimum cost path to the goal.

If this proof made sense to you, then as a studying technique to make sure you really mastered this, one thing you might consider trying is covering up the proof, and proving the theorem from scratch by yourself without referring to these notes. (This is a common studying technique for mastering proofs, and I still use it a lot when learning about new proof methods.)

4.1 Repeated states

In this section, we address a technicality regarding repeated states. As you saw in Figure 2, search trees can have repeated states, in which multiple nodes in the tree are labeled with the same state. In the most straightforward implementation of A* and other search algorithms, we would therefore end up carrying out searches from the same state multiple times, which is very inefficient.

In some problems (examples in Exercise Set 2 and in one of the questions in Problem Set 1), it is possible to formulate the search space so that there are no repeated states. I.e., each state in the discrete graph search problem is reachable only via a unique path from the initial state. In problems such as the 8-puzzle and in various “maze” or “grid search” problems (such as in Figure 3), repeated states are unavoidable, since the nature of the problem is that there are intrinsically many different paths to the same state.

Dealing with repeated states requires only a small change to our search algorithm (as stated in Section 2 and Section 3.2). Specifically, when previously the algorithm would insert a node n onto a queue, we would now modify the algorithm to consider three cases:

- If the node n is labeled with a state s that was previously expanded (i.e., if the search tree contains a previously expanded node n' that was also labeled with the same state s), then discard n and do not insert it onto the queue.

- If the node n is labeled with a state s that is already on the queue—i.e., if the queue/fringe currently contains a different node n' that is also labeled with the same state s—then keep/add on the queue whichever of n or n' has a lower priority value, and discard whichever of n or n' has a higher priority value.[3]
- If neither of the cases above hold, then insert n onto the queue normally.

If you are familiar with Dijkstra's shortest paths algorithm, you can also check for yourself that making this change to uniform cost search results in exactly Dijkstra's algorithm.

If you are implementing one of these search algorithms for a problem that does have repeated states, then making the change above to the algorithm will usually make it *much* more efficient (since now it would expand each state at most once), and this should pretty much always be done.

4.2 Monotonicity

To hone our intuitions about heuristic functions, another useful property of heuristic functions is **monotonicity**. A heuristic function h is monotonic if (a) for any nodes m and n, where n is a descendant of m in the search tree, $h(m) - h(n) \leq g(n) - g(m)$, and (b) for any goal node n_g, $h(n_g) = 0$. (The second condition is necessary, because otherwise, we could add an arbitrary constant to the heuristic function.) We can rephrase condition (a) into something more intuitive: if n is a descendant of m, then $f(n) \geq f(m)$. In other words, the heuristic function is not allowed to decrease as we follow a path from the root.

Monotonicity is a stronger assumption than admissibility, as we presently show.

Theorem 4.2: *If a heuristic function h is monotonic, then it is admissible.*

Proof: Let n_g be an arbitrary goal node, and let n be a node on the path to n_g. Then we have

$$\begin{aligned} g(n) + h(n) &= f(n) \\ &\leq f(n_g) \quad \text{by condition (a)} \\ &= g(n_g) \quad \text{by condition (b).} \end{aligned}$$

[3]For BFS and DFS which use FIFO and LIFO queues rather than a priority queue, we can discard n and keep n' on the queue.

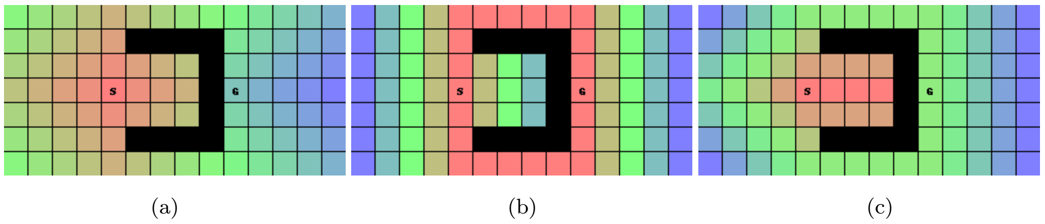


Figure 6: Some examples of the value of $f(n)$ for different nodes during A^* search, when different heuristic functions are used. The value of $f(n)$ is shown via a colorscale, where small values are red, intermediate values are green, and large values are purple. (a) $h(n) = 0$, e.g. blind search. (b) $h(n) = h^*(n)$. (c) Manhattan distance.

By subtracting $g(n)$ from both sides, we get

$$h(n) \leq g(n_g) - g(n).$$

This must hold true for any n_g which is a descendant of n, and in particular the shortest one. Hence, $h(n) \leq f^*(n) - g(n) = h^*(n)$.

The converse is not true, however, and it is possible to create admissible heuristics that are not monotonic. But this difference is largely a formal one, and in practice, admissibility and monotonicity almost always coincide. Therefore, the intuitions we develop for monotonic heuristic functions can generally be applied to admissible functions.

Monotonicity is useful, because when it holds, it implies that A^* expands all of the nodes in increasing order by $f(n)$. This follows easily from the definition; when a node n is expanded, all of its descendants (by property (a)) must have larger values of f. Hence, it is impossible for f to decrease. This gives a convenient intuitive picture of the behavior of A^* search, as shown in Figure 6. A^* first expands the red squares, followed by the orange ones, followed by the green ones, and so on.

5 Heuristic functions

Up to this point, we have treated heuristic functions as a black-box concept which we feed into our algorithms. But how do we actually come up with good heuristics? This is often highly nontrivial, and there have been many research papers published that introduce clever heuristics for given domains. Finding heuristics is hard because we are trying to balance several desiderata:

- *Admissibility.* If we are truly interested in finding the least-cost path, we want only admissible heuristics. (In practice, if we're willing to settle for a suboptimal path, we often use inadmissible heuristics to hopefully get better computational efficiency.)

- *Accuracy.* We want our heuristics to estimate the distance as accurately as possible. When working with admissible heuristics, this means we want the values to be as large as possible (but no larger than h^*).

- *Computational efficiency.* If we come up with a very predictive heuristic, it is still no good if we can't compute it efficiently. This is particularly important, since heuristic functions are usually evaluated deep in the inner loop of a search algorithm.

These different criteria lead to a tradeoff when we design heuristic functions. Two extreme (admissible) examples are the zero heuristic $h(n) = 0$ and the oracle heuristic $h(n) = h^*(n)$, which returns the actual least-cost path to the goal. The former is trivial to compute, but provides no information; using it is equivalent to blind, uniform cost search. On the other hand, if we knew $h^*(n)$, the problem would be trivial because we could head straight to the goal. Computing it, however, is as hard as solving the search problem itself. Clearly, any useful heuristic will have to lie somewhere in between these extremes. Figure 6 shows an intuitive picture of how different heuristics behave in a least-cost-path problem.

Constructing heuristics is not always easy, but there is a very useful way of thinking about the problem that often generates good heuristics. Specifically, we can often construct heuristics by thinking about solving a **relaxed problem**. In other words, we *eliminate one or more constraints* from the problem formulation. Our heuristic $h(n)$ will then be the minimum cost of getting from n to a goal, under the assumption that we do not have to obey the eliminated constraint(s). Such a heuristic will always be admissible. (Why?)

As a concrete example, recall our definition from the previous lecture of **Manhattan distance**, which measures the city-block distance from a state to the goal, *ignoring obstacles*. Basically, we eliminated the constraint that our path could not pass through an obstacle. In other words, $h(n)$ is the *exact* cost of getting from n to the goal if we were to eliminate the "you can't go through obstacles" constraint. Note that whereas computing costs $f^*(n)$ in the original problem was hard (and required performing a search), computing optimal costs when this constraint has been eliminated is easy. Analogously,

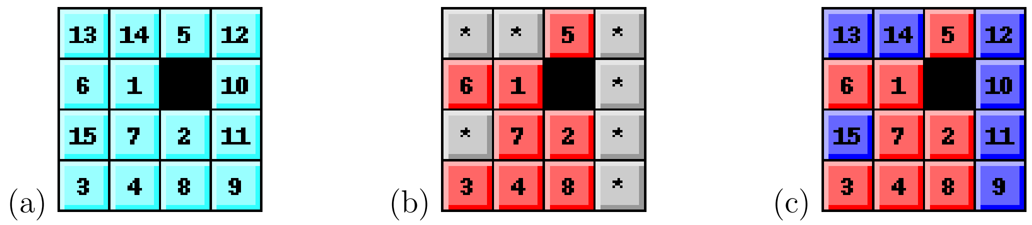


Figure 7: The heuristic for the 15-puzzle, defined as the number of moves required to move a subset of pieces into the correct positions. The number of possibilities to consider is small enough that they can all be cached in memory. (a) A possible board configuration. (b) The subset of tiles considered to compute the heuristic. Asterisks denote "don't care" tiles for the purpose of computing this heuristic. (c) An even better heuristic is if we take the maximum of the times required to get either the red tiles or the blue tiles into their goal positions.

if we are searching in a continuous space, eliminating the constraint that we can't pass through obstacles gives us the **Euclidean distance** heuristic.

More generally, even if we have an NP-hard or combinatorially large problem, eliminating one or more constraints often makes the problem trivial to solve exactly, which allows us to compute a heuristic value very quickly. For a more involved example, let us turn to the 15-puzzle (the 4x4 version of the 8-puzzle from last lecture). Recall that we can move a tile from A to B if (i) A and B are adjacent, and (ii) B is blank. Here are two examples of heuristics we can generate by eliminating constraints:

- If we eliminate (ii), we get the total number of moves the tiles have to make, if we could move them all independently without any tile getting into another tile's way. In other words, we get the sum of the Manhattan distances of all of the tiles to their final locations.

- If we eliminate both (i) and (ii), we get the number of tiles out of place. Note that, because we've eliminated more constraints, the value of this heuristic will be strictly smaller than the previous one. Since the heuristics are admissible, this is equivalent to saying it is always less accurate.

In the case of the 15-puzzle, it turns out to be possible to create an even better heuristic than these. Specifically, we will choose a subset of tiles, and pre-compute the minimum number of moves needed to get this subset of tiles from any initial configuration to their goal states. Our heuristic will ignore all the tiles except for this subset. (Thus, it corresponds to eliminating the constraint that the tiles not in this subset must also be moved to the goal state.) This is illustrated in Figure 7. Unlike Manhattan distance, this heuristic takes into account interactions between tiles—i.e., that you might need to get one of the tiles (in the subset) "out of the way" before you can move a different tile to its goal. Note also that, no matter which subset of the squares we choose to compute this heuristic on, we get an admissible heuristic. Thus, we can actually partition the tiles into two subsets, say the red and blue subsets show in Figure 7c, and compute a heuristic value for the red subset, and a heuristic value for the blue subset. In other words, we pre-compute the minimum number of moves needed to get the red tiles into their goal positions (ignoring all other tiles); and we also pre-compute the minimum number of moves needed to get the blue tiles into their goal positions. The final heuristic value that we'll give to A^* is then the *maximum* of these two values.[4] (Why not the sum? Why not the minimum?) This approach, called **pattern databases heuristics**, works very well for the 15- (and 24-) puzzle. It also gave the first algorithm that could find optimal solutions to randomly generated Rubik's cube puzzles. (In the Rubik's cube problem, we would precompute the minimum number of moves needed to get different subsets of the Rubik's cube facets into the goal positions).

Note the tradeoff here between the cost of computing the heuristic function and how much it saves us in the number of nodes we need to search. The approach described above is implemented by generating all possible board positions where only the selected subset of tiles (say the red tiles) are treated as distinct, finding the shortest paths for all configurations of this subset of tiles to their goal positions,[5] and caching the results. By including larger subsets of the tiles in this set, we could make the heuristic more accurate (closer to h^*), but this would increase the computational cost of computing the heuristic, as well as the memory requirements of storing it. We could also use a smaller subset of tiles, which would give a less accurate heuristic, but be cheaper to compute and store. More generally, there is often a tradeoff, where we can either find a more expensive, better heuristic (which would cause A^* to need to perform fewer node expansions before finding the goal), or use a heuristic that's cheaper to compute (but result in A^* needing to do more of the work in the sense of expanding more nodes). The heuristics $h(n) = 0$ and $h(n) = h^*(n)$ represent two extremes of this tradeoff, and it'll

[4]More generally, for other search problem as well, if we have several admissible heuristics, we can take the maximum to get an even better admissible heuristic.

[5]It turns out this can be done efficiently by running one big breadth first search algorithm backwards from the goal state.

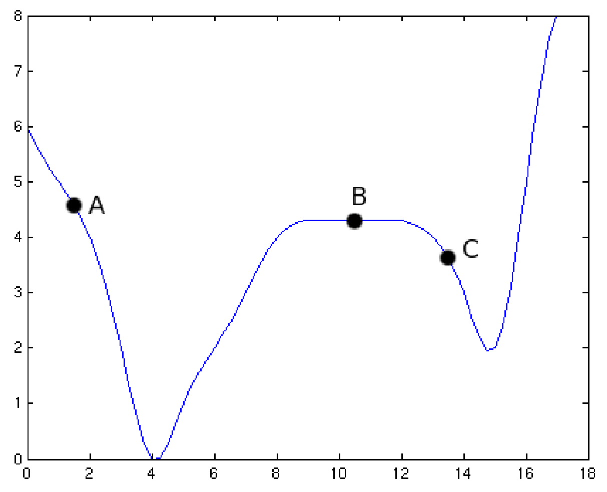


Figure 8: A cartoon example of greedy hill-climbing. The horizontal axis corresponds to the set of states. Starting from point A, the algorithm would take tiny steps downhill until it reaches the global minimum. Starting from the plateau at B, it would not know where to go because the terrain is flat. Finally, starting from C, it would move downhill and get stuck in a local optimum.

usually be something in-between that performs best.

6 Greedy hill-climbing search

Best-first search, or greedy search, expanded nodes in order of the heuristic $h(n)$. It was not optimal, because it was too greedy. Actually, it turns out that there is an even greedier algorithm, that's even more aggressive in terms of trying to get to the goal as quickly as possible. We now describe **greedy hill-climbing search** (called hill-climbing in the textbook). This algorithm throws caution to the winds and simply tries to minimize h and head straight to the goal as quickly as possible, with *no backtracking*. In other words, from a state s, we will choose whichever successor has the lowest heuristic value, and simply "move" there. Figure 8 shows a cartoon example of greedy hill-climbing search. (Given that we're actually going downhill, perhaps this is better called "hill-descending" search, but the terminology we're using is quite standard.)

Note the difference between best-first search and greedy hill-climbing search: while best-first search backtracks, greedy hill-climbing search does not. In other words, the search tree is really more of a long “chain”-structured graph. Generally, we don’t even think of greedy hill-climbing as building a tree; rather, we think of it as just hopping from one state to another. The fact that the algorithm only has to “know” about the current state and its immediate successors is often referred to as **locality**. This property gives the algorithm great power, since it can be used even when some of our earlier assumptions for path-finding are violated. For example, it works for robot motion planning even if we don’t know the locations of all of the obstacles in advance, or if the obstacles can move around.

There are two major problems commonly associated with hill-climbing search:

- **Local optima**. This is where we get to a state where all of the possible moves increase the heuristic function (make it worse). Because the algorithm doesn’t backtrack, it gets stuck. Because of local optima, hill-climbing search is usually not complete.

- **Plateaus.** The algorithm reaches a plateau when a move in any direction causes the heuristic function to remain the same. Thus, the heuristic offers no information locally about what’s a good direction to head in, and we may wander around for a long time on a large plateau before we get lucky and find a way downhill again. As a concrete example, suppose that in the 15-puzzle, we define $h(n)$ to be the number of tiles out of place. There will then often be no single move (or even any short sequence of moves) which changes the value of h. Thus, the heuristic offers no indication of where to go next, and it’s hard to make progress towards the goal.

A cartoon illustration of these problems is shown in Figure 8.

Don’t be fooled by Figure 8 into thinking that plateaus are easy to deal with. Many local search problems can involve search spaces in thousands of dimensions, and therefore plateaus can be exponentially large. Even if the plateau does have a way out, it might take so long to explore the plateau that we never find it.

6.1 Potential fields

Since we’re using greedy hill-climbing search, we’ve already given up on optimality. Therefore, there’s no need to make sure the heuristic h is admissible. In fact, there’s no need to even choose h to be an estimate of the cost to the goal—it can be anything you want, so long as going downhill on h tends to move you closer to the goal.

In motion planning, there's a particularly effective class of methods called **potential fields**, pioneered by Oussama Khatib. Concretely, we will define an **attractive** potential (in configuration space) that tends to pull us towards the goal, and a set of **repulsive** potentials that tend to push us away from the obstacles. Our robot will then perform local greedy hill-climbing search on an overall potential function that's given by the sum of all of these attractive and repulsive potentials, to try to get to areas of lower and lower potential, and hopefully all the way to the goal. An example of this is given in Figure 9.

Potential fields are extremely easy to implement, and are very space efficient because they don't need to maintain a priority queue. Furthermore, while even A^* eventually faces exponential blowup for large enough problem sizes, potential fields can sometimes scale up to very large problem sizes, because all we need to keep track of is one current state. Finally, it applies even when the obstacles may move and we don't in advance how or where they'll move. For example, in class we saw a robot soccer demo, implemented with potential fields, where the robot was attracted to the ball, and repulsed by the opposing team's robots. Even though we don't know where the opponent robots will move next, at any instant in time, all we have to do is place a repulsive potential around where they are currently, place an attractive potential centered on the ball, and take a local greedy hill-climbing step; this will tend to cause our robot move towards the ball while avoiding the other robots.

We also saw in the soccer demo that by adding other terms to the overall potential field, we could quite easily get the robots to exhibit fairly complex behavior. For example, we could use an attractive potential to the ball that's strongest for the player nearest it (so that essentially only one player at a time goes after the ball); we also saw "player position fields" that cause the players to be attracted to different areas of the soccer field, that correspond to various soccer positions, such as defender, right field, center, forward, left field; we also had a "kicking potential field" that causes the robot to be attracted to the appropriate side of the ball so as to kick it towards the opponent goal (and not towards our own goal); and so on. The combination of these relatively simple potential fields led to very rich, effective, soccer playing behavior, with each robot automatically taking into account many factors such as where the ball is, where the teammates are, where the opponent is, etc., and coordinating with its teammates to play soccer impressively well.

One disadvantage of potential fields is that, like other hill climbing search algorithms, we can get stuck in local optima. Sometimes, you might code up

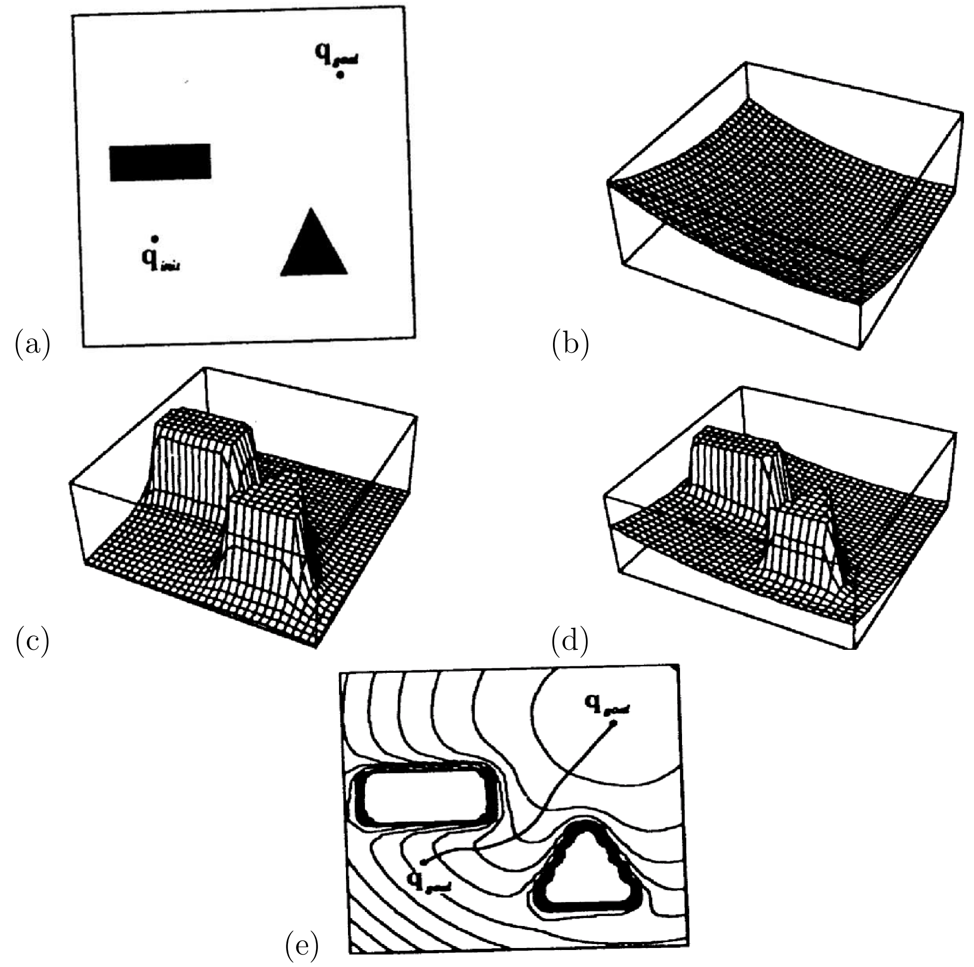


Figure 9: An example of potential fields. (a) The configuration space in a path planning problem. (b) The attractive potential which pulls the robot towards the goal. (c) The repulsive potential which keeps the robot away from obstacles. (d) The total potential. (e) The path the robot winds up following. Images courtesy of Jean-Claude Latombe.

a potential field, hand-tune it slightly and it'll give you exactly the behavior you wanted (as in the soccer example). Other times, it's also possible that if the robot keeps getting stuck in local optima (or keeps running into some other problem), and you may end up spending weeks tuning the potential functions by hand (fiddling with the strength of the attractive vs. repulsive potentials, etc.) and still not get the robot to do what you want. Thus, one disadvantage of potential fields is that if an implementation using a complex combination of potential fields does not work, it can sometimes be extremely difficult to figure out how to modify things to make it work.

Nonetheless, their scalability to huge problems, and ease of implementation, makes potential fields a very good choice for many problems. In class, we also saw a hybrid application of potential fields (together with PRMs) to plan the motions for a character in a computer graphics animation (this was the video of an animated character playing chess with a robot). That example had an extremely large state space—it used 64 degree of freedom, with each axis discretized to 100 values—but was still amenable to a potential fields solution.

6.2 Optimization search

Up until now, we have been considering a search formulation where we are interested in finding a low-cost path from the initial state to a goal. We now describe a different class of search problems, where we don't care about the cost of the path we take, but only about the final state we wind up in.

In an **optimization search** problem, we have some set of states that we search over. Furthermore, we have a cost function we want to minimize (or a goodness function we want to maximize). Usually the states will all correspond to valid solutions to some problem, and our goal is to find a good solution; in other words, a state with minimum possible cost. The solution is therefore defined *implicitly* as the lowest-cost state. In fact, sometimes it's possible for an algorithm to have already found the minimum cost state, but to have no way to realize it. Furthermore, we also don't care about the path we take to get to this state; we care only about the final state we find.

Local search methods work quite well for many optimization search problems, especially in cases where we are willing to settle for a "pretty good" (in the sense of minimizing cost), but not necessarily optimal, solution. Let us consider some examples.

6.2.1 Traveling salesperson problem

The traveling salesperson problem (TSP) is one of the classic NP-hard problems. Consider a salesperson who starts at a given city, and has a list of cities to visit. His task is to visit all of these cities, and then return to the starting city. Furthermore, he is a dishonest salesperson, and therefore can't return to the same city twice, lest he face his angry customers. The problem is to find a circuit of these cities, or **tour**, that has minimum cost.

More formally, we are given a graph G, which we assume is complete (but may possibly have edges with infinite cost). To each of the edges in G is assigned a cost. We are interested in finding the minimum-cost tour that visits each vertex exactly once, and returns to the city where we started. Since TSP is NP-hard, we shouldn't expect to find a polynomial time algorithm which guarantees finding the optimal solution. Nevertheless, local search methods often work well for finding good solutions.

Note that this problem can be expressed as a path planning problem and solved exactly using A^* search. More specifically, the states would correspond to partial tours, and each operator would extend the tour by one city, with the cost function giving the cost of the edge we just added. It is even possible to even define good admissible heuristics. Since A^* is optimal, this will eventually find the optimal solution. Since the problem is NP-hard, however, this suggests that A^* will be exponential in the worst case. In practice, we don't necessarily need to find the *best* tour, but would rather get a good tour quickly. We will use greedy local search to do this, by posing an optimization search problem.

To formulate the TSP as an optimization search problem, we need to define states, operators, and a cost function. There are many possible formulations, but here is one:

- *States:* complete tours (that visit all the cities).
- *Operators:* taking two adjacent cities in the tour and flipping the order in which they are visited.
- *Cost:* the sum of the edge costs of the edges in the tour.

Applying greedy hill-climbing search in this search space to minimize the cost function gives a nice intuitive picture, where the tour is incrementally tweaked (swapping two cities) until it gets to a local optimum—a point where the solution can't be made better by swapping two more cities.

6.2.2 Other examples

There are many other problems that can be posed as optimization search.

- **Machine Learning.** Many machine learning algorithms are formulated by posing an optimization problem, and solving it. For example, suppose we would like to have a learning algorithm estimate the prices of various houses, based on properties of the houses such as its size, etc. In the **linear regression** model (which we'll see in detail in about a week), we will write down a cost function that corresponds to how accurate our algorithm's estimate of housing prices are. We then apply a greedy hill-climbing search algorithm to minimize the errors in our algorithm's predictions.

- **8 queens.** One famous toy problem is the 8 queens puzzle. In the game of chess, a queen can attack any piece which shares a row, column, or diagonal with it. The goal is to place 8 queens on an 8×8 chessboard such that no queen can attack any other queen. Local search methods have been shown to work surprisingly well for this problem. In one possible formulation, each state corresponds to an assignment of all of the queens to squares on the chess board, and the cost function is defined as the number of queens under attack. Our operators can take one of the queens, and move it elsewhere on the board.

- **Building an automobile.** In order to build a car in an automobile plant, we must decide on the order in which parts will be fabricated, finished, painted, combined, and so on. We must schedule jobs on different machines, each of which needs to work on different parts for a different amount of time. The aim is to maximize the output of a factory, so this is an optimization search problem. By applying simulated annealing (a kind of local search) to the car design problem, an engineer at General Motors found a way to save $20 per automobile.

6.3 Propositional satisfiability

One problem which has received a huge amount of attention is **propositional satisfiability**, also known as **SAT**. This is an NP-hard problem — in fact, the original NP-hard problem — but can often be solved efficiently in practice. Here, we are given a sentence in **propositional logic**; in other words, a sentence built out of a set of propositional variables $A_1, \ldots, A_n$, each of which can be either true or false, and the propositional connectives

$\wedge$ (and), $\vee$ (or), and $\neg$ (not). For example, the following sentence is true if and only if A and B are either both true or both false:

$$(A \wedge B) \vee (\neg A \wedge \neg B)$$

Given a sentence in propositional logic, our goal is to determine if there is a **satisfying assignment**, meaning an assignment of true or false to each of the variables which causes the sentence to evaluate to true.

SAT is an important problem because many other problems in AI, such as planning, can be solved by turning it into a huge SAT problem. In fact, one effective way to solve many other AI problems is by converting them into a big SAT problem, and feeding it to a state-of-the-art SAT solver. This takes advantage of the immense effort which has already been spent engineering efficient SAT solvers, and has the added benefit that we get a free performance boost when newer and better SAT solvers are released.

When we discuss SAT algorithms, we consider a particular type of propositional sentence called **conjunctive normal form** (CNF). This form is popular because it significantly simplifies the process of designing SAT algorithms, yet there exists a straightforward procedure to translate any propositional sentence into CNF.

To define CNF, we first introduce some terminology:

- A **literal** is a single variable, possible negated. For example, A_3 or $\neg A_5$.
- A **clause** is a disjunction ("or") of literals. In other words, it is true if any one of the literals is true. For example, $A_3 \vee \neg A_7 \vee A_8$.

A CNF sentence is a conjunction ("and") of clauses. Therefore, a CNF sentence evaluates to true if and only if at least one of the literals is satisfied in each clause. Here is an example CNF sentence:

$$(A_1 \vee A_3 \vee \neg A_4) \wedge (\neg A_1 \vee A_2 \vee A_3) \wedge A_5$$

Since our goal is for the number of unsatisfied clauses to equal zero, we can formulate it as a local search problem in the following way:

- *States:* complete assignments to all of the variables
- *Operators:* flipping the truth value of a single variable
- *Cost:* the number of unsatisfied clauses

Greedy hill-climbing search for SAT, also known as GSAT, iteratively flips the variable which causes the greatest net gain in the number of satisfied clauses. Ties are broken randomly. Despite its simplicity, GSAT is a surprisingly effective SAT algorithm. There're also many other algorithms (such as WalkSAT) which implement various improvements to GSAT, and work even better. The current state-of-the-art can solve often problems with tens of thousands of variables, which is quite impressive for an NP-hard problem.

Additional notes

- **Figure 1:**
  states: integer locations of tiles (ignore intermediate positions)
  actions: move blank left, right, up, down (ignore unjamming etc.)
  goal test: = goal state (given)
  path cost: 1 per move
- Optimal solution of n-Puzzle family is NP-hard]
- **State vs node**

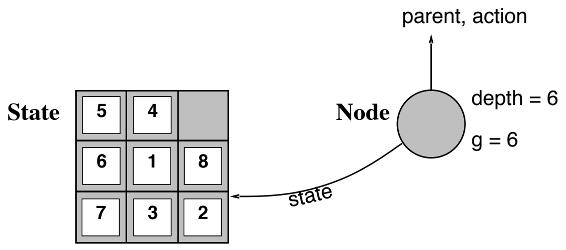
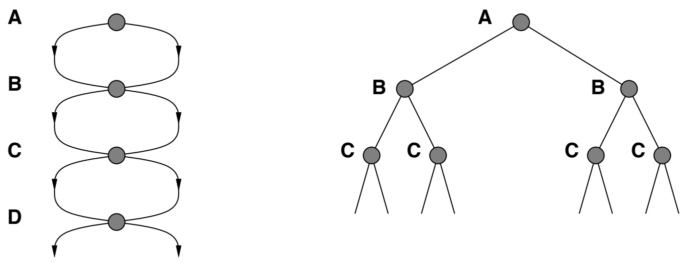


Fig. 3.1 States vs. nodes: A *state* is a (representation of) a physical configuration, a *node* is a data structure constituting part of a search tree. A search graph includes *parent*, *children*, *depth*, *path cost* $g(x)$. States do not have parents, children, depth, or path cost!

- **Strategies are evaluated along the following dimensions:**
  - *completeness:* does it always find a solution if one exists?
  - *time complexity:* number of nodes generated/expanded
  - *space complexity:* maximum number of nodes in memory
  - *optimality:* does it always find a least-cost solution?
- **Repeated states:** Failure to detect repeated states can turn a linear problem into an exponential one!



- **More search algorithms:**
  - bi-directional search
  - see `http://en.wikipedia.org/wiki/Graph_traversal`

Constraint satisfaction problem (CSP)

4

4.1 CS221 Lecture notes by Andrew Ng, No. 4

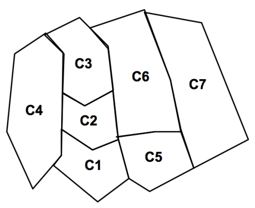
CS221 Lecture notes #4

Constraint satisfaction problems (CSP)

In the previous set of notes, we discussed the application of local search algorithms to problems such TSP, 8-queens and SAT. In those problems, we were interested in finding a solution, but we didn't care how we get to a good state. Greedy hill-climbing search is an effective algorithm for these problems, but for a subset of them, we will be able to devise more efficient search algorithms, ones that rely on our making explicit more of the structure in the problem.

Recall that by incorporating a heuristic function, we converted uniform cost search into a much better algorithm, A^*. The heuristic function was a way to convey information about the problem to the algorithm, so that A^* could reason about what nodes it could skip expanding, and still guarantee finding an optimal solution. That reasoning process was an example of **inference**.

In today's lecture, we'll describe a formalism that makes more of the structure of certain search problems explicit to the search algorithm, so that it can reason more "deeply" into the problem and perform deeper inference regarding the space of solutions. Specifically, we will describe **constraint satisfaction problems (CSP)**, where the goal is to find assignments to a set of variables, so that the assignments satisfy a certain set of constraints. The CSP problem specification will allow us to do inference to prune off huge portions of the search space which would have been explored by simpler algorithms such as blind search. In some cases, it will also help us to choose the most promising areas of the state space to explore first.

(a)


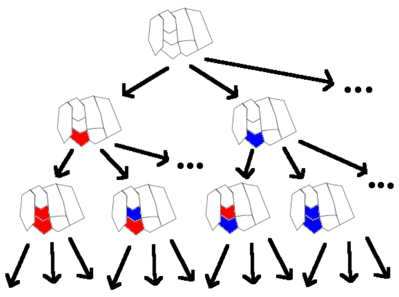
(b)


Figure 1: (a) A map which we want to color using only three colors, with no two adjacent countries sharing the same color. (b) A portion of the search tree if we attempt to solve the problem using naive depth-first search. Note that the search continues down the left hand side even though the color constraint is already violated after just two steps.

1 Problem Definition

First, a motivating example. Consider the problem of coloring the map in Figure 1 with no more than three colors, such that no two adjacent countries share the same color. How might we solve this problem *naively* using depth-first search? We define a variable C_i for each region, which can take values in the set {red, green, blue}. The states will be partial assignments of values (colors) to the variables. The operator takes the next uncolored region, and colors it. The goal test takes a full assignment of colors to all of the countries and checks if all pairs of adjacent countries have different colors.

What happens when we apply naive depth first search (DFS) using this formalism? Part of the search tree is shown in Figure 1. The algorithm first proceeds down the left-hand side of the search tree, assuming the first operator is to color the next country red. Therefore, it immediately colors countries 1 and 2 red. This seems silly to us, because we know there can't possibly be a satisfying solution where both of these countries are red. But remember that for general search, the goal test is simply a black box which takes a state and tells us if it is a goal. It can't tell us whether or not a state might lead to a goal later on, and so after having colored countries 1 and 2 red, it will waste a large amount of time exploring all possible combinations of colors for countries 3, 4, . . . , 7, not realizing that there's no combination of colors for them that could lead to a solution in this leftmost portion of the tree where countries 1 and 2 have already been colored red.

In order to make this sort of information available to the algorithm, we must make the constraints explicit, or **declarative**. This will allow us to construct algorithms which can prune large portions of the search space as soon as inconsistencies are introduced.

A **constraint satisfaction problem** (CSP) comprises the following:

- A set $V = \{v_1, \ldots, v_n\}$ of variables. In our example, V is the set of countries.
- A finite domain $D = \{d_1, \ldots, d_m\}$ of possible values for each variable. (When the domain is different for different variables, we will use D_i to denote the domain of variable V_i.) In our example, D is the set {red, green, blue}.
- A set C of constraints. Each constraint defines some restriction on the possible values a subset of the variables may jointly take. Each constraint is defined by specifying a subset $V' \subseteq V$ of variables, and the set of legal values (tuples) for the variables in V'. For example, the constraint that neighboring countries have different colors can be represented as:

$$\{(\text{red, green}), (\text{red, blue}), (\text{green, red}),$$
$$(\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$$

Our goal is to find a set of domain values to assign to all the variables V_i. so that all the constraints in C are satisfied simultaneously. Let us look now at some additional examples of CSPs.

1.1 8 queens problem

Recall that, in chess, a queen can attack any piece which lies on the same row, column, or diagonal. We want to place 8 queens on a chess board such that no one queen can attack any other queen. An example solution to the 8 queens problem is shown in Figure 2.

There are many ways to formulate the problem of finding a solution to the 8 queens problem as a CSP. Here is a fairly good one, which uses the insight that there has to be exactly one queen per column:

- **Variables:** V_i for $i = 1, \ldots 8$, representing the row number of the queen in the i^{th} column.
- **Domain:** $\{1, \ldots, 8\}$

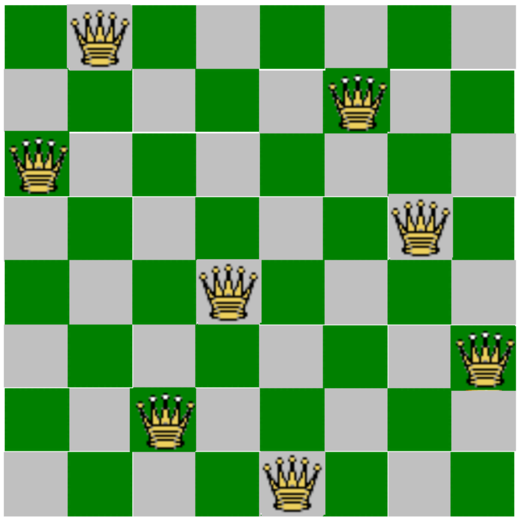


Figure 2: An example of a solution to the 8 queens puzzle.

- **Constraints:** For each V_i and V_j, the constraint that they cannot attack each other. For each V_i and V_j, (r_i, r_j) is a legal pair of values if:

$$\begin{aligned} &r_i \neq r_j \quad \text{for } i \neq j \\ &|r_i - r_j| \neq |i - j| \quad \text{for } i \neq j \end{aligned}$$

  The first line above ensures that the queens in columns i and j aren't in the same row ($r_i \neq r_j$). The second line above ensures that the queens in columns i and j aren't along the same diagonal. (This is perhaps an unfortunately cryptic way of writing it, but it is basically the constraint that the difference in x-coordinate $|r_i - r_j|$ isn't the same as the difference in y-coordinate $|i - j|$—in other words, that they aren't on the same diagonal.)

1.2 Scheduling Ph.D. visits

Here's our third example. It's visiting weekend for Stanford Ph.D. admits, and there are n admitted students who want to meet with all n professors. All these meetings must happen in the n available time slots; further, each person can be in only one meeting at a time, so if a student and a professor are meeting at some time, then neither one of them can have any other meeting at the same time. We can represent this as a CSP:

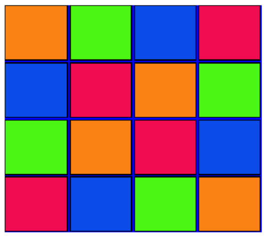


Figure 3: An example of a completed Latin square.

- **Variables:** V_{ij} for each student i and professor j, giving the time slot in which they meet.
- **Domain:** $\{1, \ldots, n\}$
- **Constraints:** For all i, j, $i' \neq i$, and $j' \neq j$, we have $V_{ij} \neq V_{i'j}$ and $V_{ij} \neq V_{ij'}$. In other words, no student or professor can be in more than one meeting at a time.

We can also represent this problem pictorially. Imagine a matrix where the rows and columns represent students and professors respectively, and the value in each cell represents the meeting time. If we use a different color to represent each of the n possible timeslots, the problem is to find a way to color the cells such that each color appears only once in each row and column. This is also known as the **Latin square** problem, and an example solution is shown in Figure 3. Often, we are interested in solving a harder problem: given an assignment of colors to a subset of the cells, is it possible to find a coloring of the remaining cells which gives a Latin square?

It turns out that, more generally, most scheduling problems can be formulated as CSPs.[1]

2 Inference algorithms

Now that we've defined the constraint satisfaction problem, how do we actually solve it? Actually, we are not going to present individual monolithic

[1]For example, in case our Ph.D. visit weekend scenario seems contrived, note that n students, each of whom wants to meet a different subset of m professors, in a total of t timeslots (and with each professor being available for a different subset of the timeslots), is also a CSP, and this is what actually happens at Stanford's Ph.D. visit weekend. This more general formulation doesn't lead to the nice Latin square solutions, though.

algorithms for solving CSPs. Rather, we will present three **inference procedures** of increasing sophistication, and then we will discuss a variety of heuristics which further help in the search process. Our inference procedures will allow us to rule out large sections of the search tree by proving it could not possibly contain a solution. These inference procedures and heuristics will be embedded in the overall search algorithm, and can be mixed and matched in various ways depending on the task at hand.

Recall our description previously of a depth-first search process over a state space of partial assignments to variables. Our operators in the search space will take one unassigned variable, and assign a value to it. This depth-first search algorithm will serve as our overall template. Now, however, we will check at each node which assignments could possibly lead to solutions. This will allow us to select more promising assignments for each variable, or to backtrack earlier when it is clear no assignment will work.

2.1 Consistency checking

When we discussed the map coloring example, we saw that naive depth-first search wasted time considering an entire section of the search tree even though it had already assigned two neighboring countries the same color. The most basic inference algorithm, **consistency checking**, simply rules out this case. Specifically, we say that a partial assignment is **consistent** if it does not already violate any of the constraints in C.[2] In consistency checking, we simply don't assign any value v_i to V_i which leads to an inconsistent partial assignment. For instance, in map coloring, we don't even consider coloring C_2 red if we have already colored C_1 red. As shown in Figure 4, this often allows us to backtrack early.

2.2 Forward checking

A somewhat more powerful inference algorithm is called **forward checking**. We don't have to wait until we assign a value v_j to variable V_j to check which assignments are consistent. Rather, each time we instantiate a variable V_i, we can propagate all of its constraints forward. For each variable V_j which has not been instantiated, we remove from its domain all values which conflict with V_i. An example is shown in Figure 5. The key advantage of forward

[2]Technically, this definition only works for binary constraints (constraints between two variables). For N-ary constraints with $N > 2$, an assignment to a subset of the variables in a constraint C is consistent if there exists some tuple in C which does not contradict that assignment.

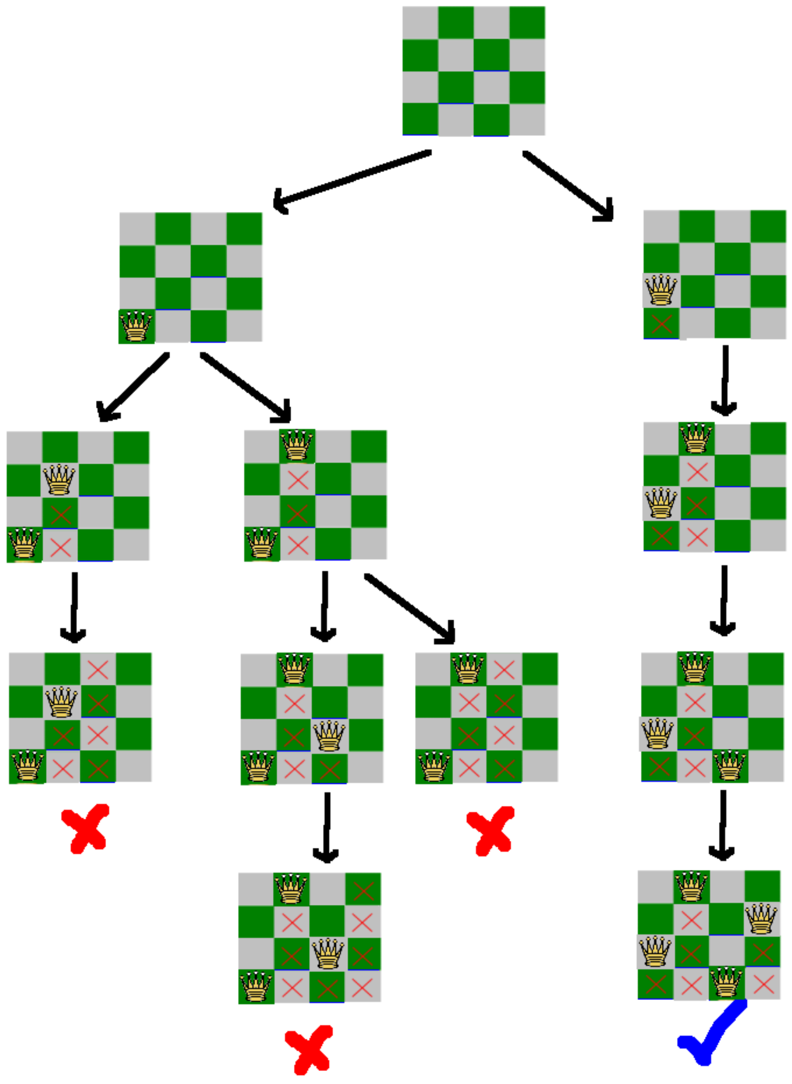


Figure 4: The complete search tree for solving the 4 queens problem using consistency checking.

checking over consistency checking is that it allows us to detect early on when a given variable has no possible consistent assignments, and therefore backtrack. A case where this makes a difference is demonstrated in Figure 6. When we only use consistency checking, we have to wait until the offending variable V_6 is actually instantiated.

2.3 Arc consistency and constraint propagation

The most powerful inference algorithm we will discuss here, **constraint propagation**, actually considers pairs of uninstantiated variables and rules out inconsistent pairings. The goal is to eliminate values from domains until all of the domains become arc consistent, a term we define shortly. This often rules out significantly more possibilities than forward checking, but at a much larger computational cost. For simplicity, let's restrict our attention to binary constraints (those involving only two variables).

For intuition, consider a pair of variables V_i and V_j, with domains D_i and D_j. (These are not the original domains from the problem definition, but rather the domains with some values already removed by constraint propagation.) Suppose we try all of the assignments $v_j \in D_j$ for V_j, and all of them turn out to be inconsistent with some specific assignment v_i to V_i. We can then rule out assigning v_i to V_i, since it is inconsistent with all remaining values for V_j.

Definition. Let c be a constraint over two variables V_i and V_j. A value v_i for V_i is **c-consistent** with D_j if there is some possible assignment $v_j \in D_j$ to V_j which does not violate c (more formally, $(v_i, v_j) \in c$). A domain D_i is c-consistent with a domain D_j if all values $v_i \in D_i$ are c-consistent with D_j. If D_i is c-consistent with D_j, we also say that D_i is **arc consistent** with D_j.[3]

The basic step in our algorithm is **constraint propagation**. Given a constraint c over variables V_i and V_j, we **propagate** c from j to i by eliminating from D_i all values v_i that are not c-consistent with D_j. After this step, D_i will be c-consistent with D_j.

The goal of arc consistency checking is to ensure that all pairs of domains are arc-consistent. To ensure arc consistency, on each pass over the variables, we perform constraint propagation on all pairs V_i, V_j of variables for $i \neq j$. Note that our definitions of c-consistency and constraint propagation

[3]For general CSPs, two domains D_i and D_j can be included in multiple constraints, and in this case, c-consistency and arc consistency are different. Here, however, we restrict ourselves to the case of binary constraints, so we treat the two as the same.

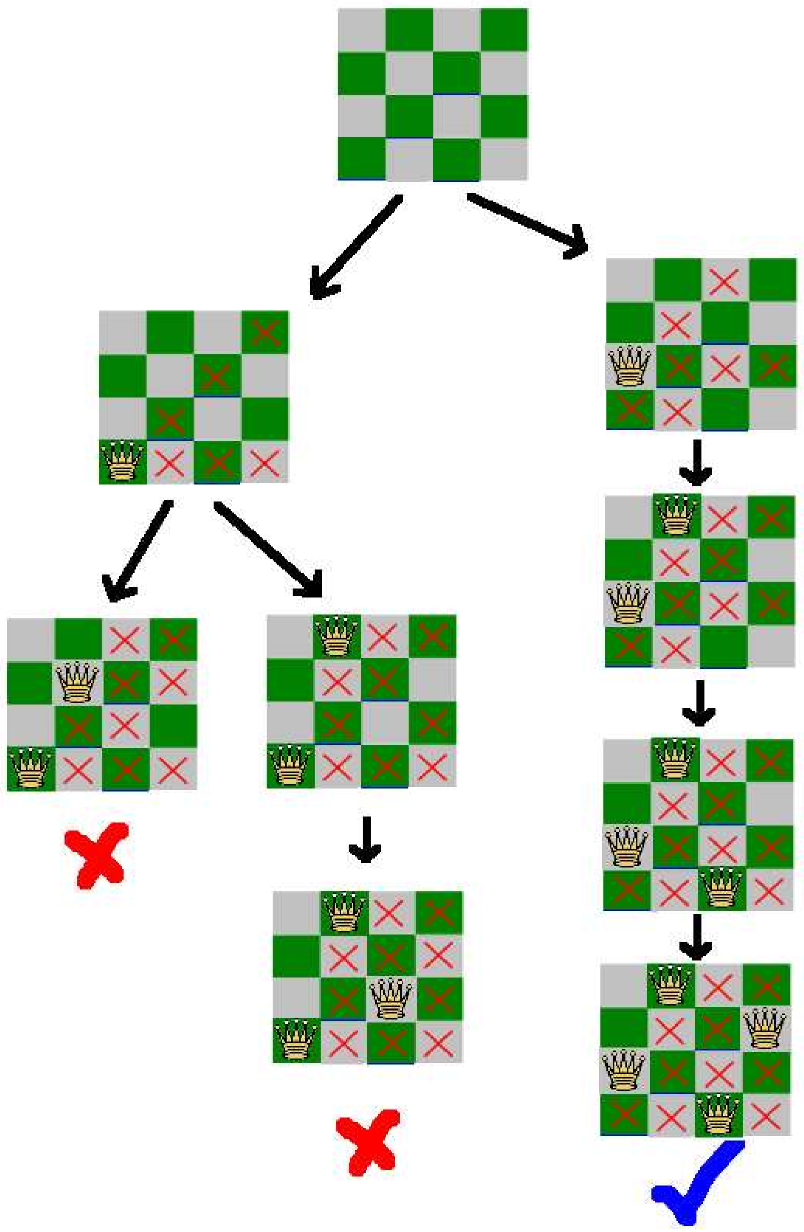


Figure 5: The complete search tree for solving the 4 queens problem using forward checking.

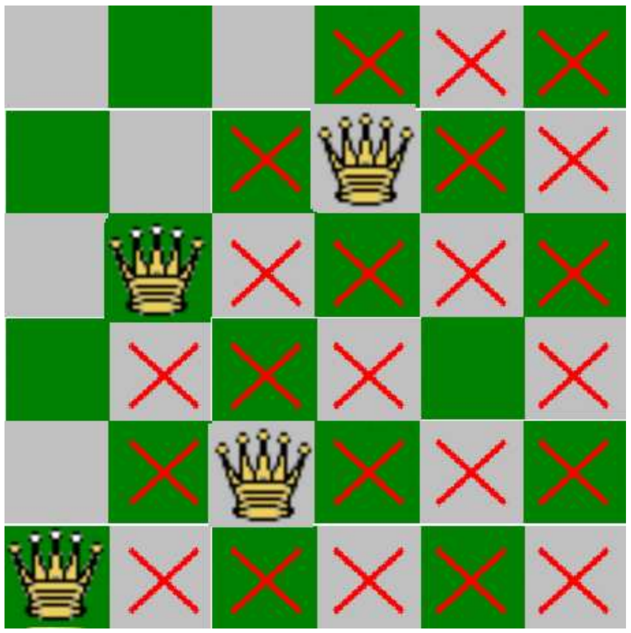


Figure 6: An example of a search node in the 6 queens puzzle where forward checking saves a lot of time relative to consistency checking. As soon as we instantiate the fourth queen, we detect that the sixth queen has no moves left, and we can backtrack. Using consistency checking, we would have to wait until we instantiated the sixth queen, thereby going two levels deeper into the search tree.

are not symmetric, so we must perform constraint propagation separately on (V_i, V_j) and (V_j, V_i). Because removing a value from one domain might make it possible to remove other values from other domains, we must keep repeating the above procedure until the graph is arc consistent. (Make sure you understand why this procedure will eventually terminate and guarantee arc consistency.)

In our description of the algorithm thus far, we talked about iterating over all pairs of variables V_i, V_j. We described the procedure this way since it made it conceptually simpler to describe, but repeatedly iterating over all pairs of variables would be unnecessarily inefficient, and real implementations of arc consistency typically use somewhat more efficient ways to organize the ordering of the constraint propagation steps and keep track of what constraints need to be propagated next. (For example, one such algorithm is **AC-3**, which is described in the textbook.)

Suppose we run arc consistency to convergence, until no more domain values can be eliminated. What happens when we're done propagating constraints at a particular point in the search tree? There are three possibilities:

- For some variable V_i, we have $D_i = \emptyset$. We know then the problem is inconsistent, so we can backtrack.[4]
- For all variables V_i, we have $|D_i| = 1$. I.e., every D_i is a singleton set. In this case, we are done—we've found a solution, and can simply assign all remaining variables to the value in their corresponding domains. (Why are we guaranteed that this won't violate any additional constraints?)
- D_i has multiple remaining values. In this case, we have to keep searching. I.e., we instantiate some value for D_i, and keep searching.

In practice, full arc consistency is often too expensive to run deep in the inner loop of a search algorithm, so we usually make a compromise between effective inference and computational efficiency. For instance, one common choice is to check arc consistency only at the very highest levels of the search tree, and then use only forward checking for the rest of the search. Alternatively, we might check arc consistency only for a fixed subset of the constraints. This is the usual tradeoff between expensive inference to allow

[4]Note that these domain variables D_i (in both the cases of forward checking and constraint propagation) depend on what node of the search tree we are in. In particular, when our DFS procedure backtracks and "unassigns" some variable, we need some sort of data structure to keep track of which values were previously removed from a given domain D_i and need to be added them back as part of the backtracking step.

us to search fewer nodes, vs. using cheaper inference but more brute force to search more nodes.

3 Heuristics

In the last section, we outlined three inference algorithms to use inside the search. Even with these algorithms, however, as part of DFS we have to make two arbitrary decisions:

- Which variable to instantiate next.
- For a given variable, which values in the domain to try first.

For both of these, we have assumed a naive approach, where we arbitrarily numbered our variables $V_1, \ldots, V_n$ and the domain values $d_1, \ldots, d_n$, and we checked them in those orders. It turns out we can do significantly better using two simple heuristics. These heuristics can be used with any of the three inference algorithms presented above.

3.1 Most constrained variable

First, we tackle the question of which variable to instantiate next. Intuitively, if we are exploring the wrong part of the search tree, we want to find out as soon as possible. One way to do this is by using the **most constrained variable (MCV)** heuristic (also called the **minimum remaining values (MRV)** heuristic), in which we start by instantiating the variable with the fewest remaining values left in its domain. The fewer values we have left, the fewer we have to try before we reach a contradiction. As a special case of this heuristic, if a variable only has one value left in its domain, we may as well go ahead and assign that variable.

3.2 Least-constraining value

Once we've chosen a variable to instantiate, in what order do we try the different values? In this case, the order in which we choose values won't affect how long it takes to find a contradiction. If there is a contradiction, we will have to try all of the values no matter what. On the other hand, if we are in the correct part of the search tree, we want to find a correct solution as soon as possible. This suggests choosing the value which would remove the fewest values from other variables' domains, or the **least constraining**

value (LCV), since this leaves more of our options open, so that there's hopefully more likely for there to be a solution to be found still.

Note a key difference between these two heuristics in terms of how we treat them when backtracking. Under the LCV heuristic, if we color Country 2 green and it doesn't work, it makes sense to backtrack and try red. In contrast, choosing variable ordering is a **non-backtrack step**. E.g., if we choose to color Country 2 first and find that that doesn't work—none of the values work with our current assignment—then it doesn't make sense to go back and try coloring Country 4 first.

Part II

Machine Learning

Learning machines and the perceptron

5

This chapter gives a brief historical introduction to learning machines and neural networks, before treating this area in a more modern fashion in the following chapters.

5.1 Learning Machines

We have, so far, mainly looked at programming strategies to solve complex tasks. Specifically, we formulated complex tasks as search problems in a configuration space and then discussed general search strategies in potentially large spaces. *Search* is a huge component of AI. Of course, the real challenge is with the AI engineer to translate the problem into the programmable structure (abstractions) and to find the principle solution strategy.

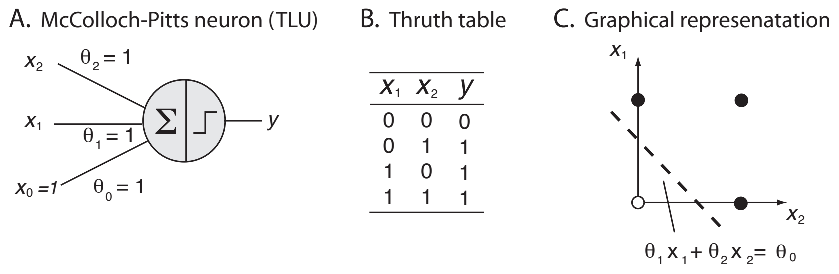
A further main area of AI is that of *reasoning*. In this area, we consider some information given to us and how we combine such information to make statements that are derived from such facts (e.g. *if such and such, then ...*). This included propositional logic and higher order logic.

Many of these strategies have gone a long way to solve complex tasks such as providing support for decision making or making computers that are good game players. However, many years of research and the analysis of common challenges has pointed to a common problem in AI. The major challenges for AI is that many problem domains in the real world are, at least in the eyes of a common observer, *ever changing and unreliable*. Modern research is AI is therefore looking into tackling this problem domain with two major strategies that dominate modern AI, that of *machine learning* (ML) and *probabilistic reasoning* (PR). Learning machines are important for several reasons. For example, such machines have the potential to find solutions strategies on their own when solutions are not known a priori, and, maybe more importantly, machines that are capable of learning have the ability to adapt to changing situations or to situations that have not been considered by the engineer when setting up the machine.

The rest of this course will focus on ML and PR. Both areas deal with uncertain environments and unreliable components such as sensors and actuators where a probabilistic description is appropriate. A probabilistic framework is therefore useful in the following treatments, and a refresher of probability theory is provided in the tutorials.

5.2 The McCulloch-Pitts neuron model

In this chapter, we briefly discuss the historical area of learning machines, specifically the area that became known as Neural Networks. There was always a strong interest of AI researchers in *real intelligence*, that is, to understand the human mind. For example, both Alan Turing and John von Neumann worked more directly on biological systems in their last years before their early deaths, and human behaviour and the brain have always been of interest to AI researchers.


A. McColloch-Pitts neuron (TLU)
x_2
$\theta_2 = 1$
x_1
$\theta_1 = 1$
$x_0 = 1$
$\theta_0 = 1$
Σ
y
B. Thruth table
C. Graphical represenatation
x_1
x_2
$\theta_1 x_1 + \theta_2 x_2 = \theta_0$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 5.1 Representation of the boolean OR function with a McCulloch-Pitts neuron (TLU).

A seminal paper, which has greatly influenced the development of early learning machines, is the 1943 paper by Warren McCulloch and Walter Pitts. In this paper, they proposed a simple model of a neuron, called the *threshold logical unit* or *McCulloch–Pitts neuron (node)*. Such a unit is shown in Fig. 5.1A with three input channels, although it could have an arbitrary number of input channels. Input values are labeled by x with a subscript for each channel. Each channel has also a *weight parameter*, θ_i. The TLU unit operates in the following way. Each input value is multiplied with the corresponding weight value, and these weighted values are then summed. If the weighted summed input is larger than a certain threshold value, θ_0, then the output is set to one, and zero otherwise, that is,

$$h_\theta(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i \theta_i x_i = \theta^T \mathbf{x} > \theta_0 \\ 0 & \text{otherwise} \end{cases} . \tag{5.1}$$

Such an operation resembles, to some extend, a neuron in that a neuron is also summing synaptic inputs and fires (has a spike in its membrane potential) when the membrane potential is higher than some level. While McCulloch and Pitts introduced this unit as a simple neuron model, But they also argued that such a unit can perform computational tasks resembling boolean logic. This is demonstrated in Fig. 5.1. The symbol h is used in these lecture notes since the output of the perceptron is the *hypothesis* of the perceptron, given the parameters θ.

5.3 The perceptron

The next major developments in this area were done by Frank Rosenblatt and his engineering colleague Charles Wightman (Fig. 5.2), using such elements to build a machine that Rosenblatt called the *perceptron*. As can be seen in the figures, they worked on a machine that can perform letter recognition, and that the machine consisted of a lot of cables, forming a network of simple, neuron-like elements.

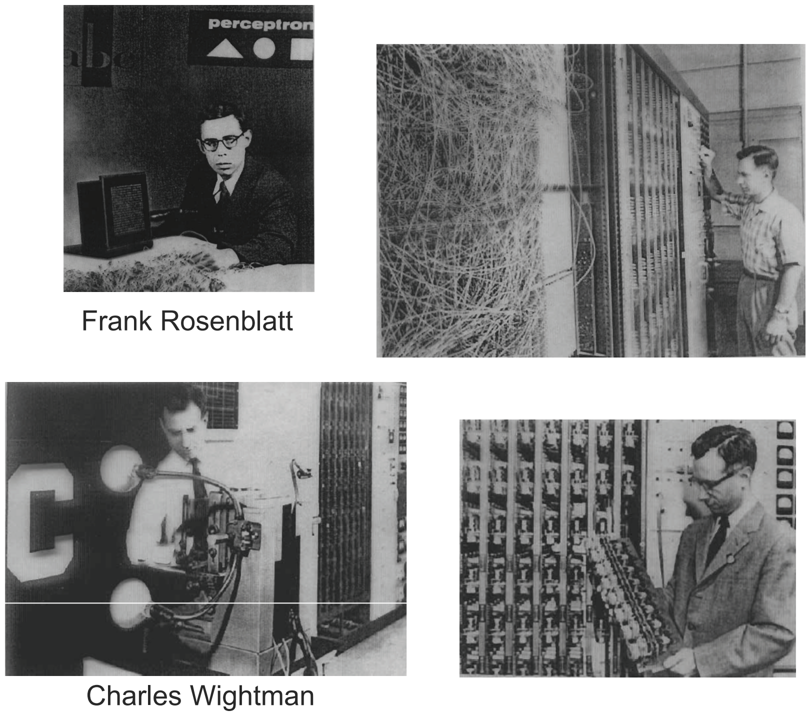


Fig. 5.2 Neural Network computers in the late 1950s.

The most important challenge for the team was to find a way how to adjust the parameters of the model, the connection weights θ_i, so that the perceptron would perform a task correctly. The procedure was to provide to the system a number of examples, let's say m input data, $\mathbf{x}^{(i)}$ and the corresponding desired outputs, $y^{(i)}$. The procedure to update the parameters of the systems based on these training examples is called a *learning rule*, and this form of learning with explicit examples is called *supervised learning.* The *perceptron learning rule* is given by

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(\mathbf{x_i}) \right) x_j^{(i)}. \tag{5.2}$$

This learning rule is also known, or related to, the Widrow-Hoff learning rule, the Adaline rule, and the Delta rule.[1] It is often called the delta-rule because the difference between the desired and actual output (difference between actual (training) data and hypothesis) to guide the learning. When multiplying out the difference with the inputs we have end up with the product of the activity between the inputs and output values for each synaptic channel. Such a learning rule is also called Hebbian after the famous NovaScotian Donald Hebb.

[1] These learning rules are nearly identical, but are sometimes used in slightly different contexts. We will not discuss these issues further.

There was a lot of excitement during the 1960s in the AI and psychology community about such learning system that resemble some brain functions. However, Minsky and Peppert showed in 1968 that such perceptrons can not represent all possible boolean functions (sometimes called the XOR problem).

While it was known at this time that this problem could be overcome by a layered structure of such perceptrons (called *multilayer perceptrons*), a learning algorithms was not widely known at this time. This killed the field, and the AI community concentrated on rule-based systems in the following years. The generalization of a delta rule, known as error-backpropagation, was finally introduced by Rumelhart, Hinton and Williams in 1992 (although Paul Werbos, and also Sunichi Amari, used it before), and resulted in the explosion of the field of *Neural Networks*. This area has now become known as machine learning, which has clarified a lot of the abilities and challenges of neural networks. We therefore follow in the next sections a more contemporary path.

5.4 Summary

The area of Neural Networks has been active since the 1950s. A large potion of this area is concerned with supervised learning, which we will discuss further in the next sections. This area is now mainly absorbed into the field of machine learning. There is also an active area modelling brain functions known as *computational neuroscience*. This area is subject of CSCI6508/NESC4177 taught in the winter term. While these fields have developed into some different directions and have some distinct goals, there is now some exciting convergence when it comes to unsupervised learning and complex modelling. This will be subject of later discussions in this course.

Regression, classification and maximum likelihood

6

6.1 CS221 Lecture notes by Andrew Ng, No.5

CS221 Lecture notes #5

Supervised learning

So far in this course, we have only considered problems where the entire state of the world is known in advance. Rarely, however, can we completely specify the state of the world a priori, so often the agent must be able to learn about the world from actual observations. For instance, we might be interested in automatically distinguishing different handwritten digits. It's hard for us to formally state a set of rules which distinguish handwritten 2's from 5's, and so we can't simply program it directly into the computer. Rather, we will have an algorithm automatically figure it out from data. The general set of techniques for doing this is known as **machine learning**. Since machine learning is such a broad field, there is no single definition that everybody agrees upon, but here are some attempts:

- Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.[1]
- A computer program is said to learn from experience E with respect to some task T and some performance measure P if its performance on T, as measured by P, improves with experience E.[2]

An example of an early machine learning program was Arthur Samuel's chess playing program, which learned to play checkers by playing many games against itself, and eventually learned to play much better than Samuel himself. Since them, machine learning has produced many practical applications.

For the next two lectures, we will focus on **supervised learning**, where our algorithm will work with *labeled* training examples, or examples which

[1]Arthur, S. "Some studies in machine learning using the game of checkers." *IBM Journal* (3): 210-229.

[2]Mitchell, T. *Machine Learning.* McGraw-Hill, 1997.

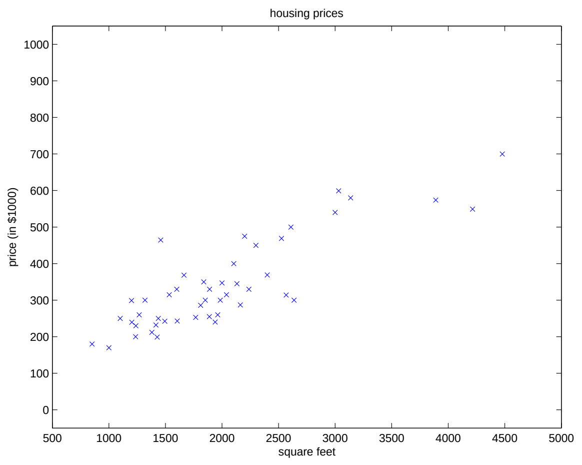
are labeled with the property we are trying to predict. For instance, if we are trying to classify handwritten digits, labeled data would constitute image files which are labeled by humans as being a particular digit.

1 Linear regression

As a motivating example, suppose we have a dataset giving the living areas and prices of 47 houses from Portland, Oregon:

Living area (feet^2)	Price (1000\$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

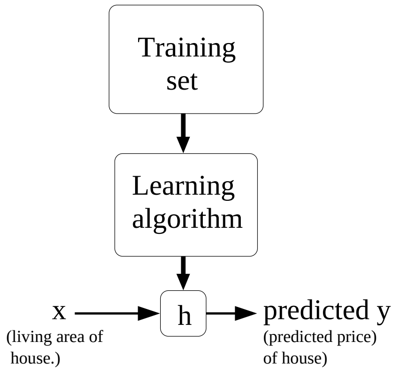
We can plot this data:



Given data like this, how can we learn to predict the prices of other houses in Portland, as a function of the size of their living areas?

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables (living area in this example), also called input **features**, and $y^{(i)}$ to denote the "output" or **target** variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$—is called a **training set**. Note that the superscript "(i)" in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use $\mathcal{X}$ denote the space of input values, and $\mathcal{Y}$ the space of output values. In this example, $\mathcal{X} = \mathcal{Y} = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : \mathcal{X} \mapsto \mathcal{Y}$ so that $h(x)$ is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a **hypothesis**. Seen pictorially, the process is therefore like this:



When the target variable that we're trying to predict is continuous, such as in our housing example, we call the learning problem a **regression** problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a **classification** problem.

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. In **linear regression**, our hypotheses are linear functions of the features. In the case of housing prices, we have one feature x representing the area, and so our hypothesis is

$$h_\theta(x) = \theta_0 + \theta_1 x.$$

(We will drop the subscript θ when there is no risk of confusion.) We might also happen to know the number of bedrooms as well. In this case, we have two features x_1 and x_2, so our hypothesis is

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2.$$

For the purposes of this lecture, our hypothesis will be a linear combination of n features, plus the intercept θ_0. For notational convenience, we introduce the convention of letting $x_0 = 1$. Then, we can rewrite our hypothesis as:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i.$$

Equivalently, we can rewrite this using an inner product:

$$h(x) = \theta^T x,$$

where on the right-hand side we are viewing θ and x both as vectors in $\mathbb{R}^{n+1}$, and n is the number of input variables (not counting x_0). We refer to θ as the **parameters**, or **weights**, of the hypothesis. θ_0 is also referred to as the **intercept term**.

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method seems to be to make $h(x)$ close to y, at least for the training examples we have. To formalize this, we will define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

In other words, we penalize the squared magnitude of the **error term** $(h_\theta(x^{(i)}) - y^{(i)})^2$ for each training example. This algorithm is commonly referred to as **ordinary least-squares**.

2 Gradient descent

We want to choose θ so as to minimize $J(\theta)$. This is an optimization search problem, since we are only interested in finding a good value of θ, and we don't care about the path we take to find it. This suggests using greedy hill-climbing search. In our previous formulation, we applied a fixed set of operators to generate all of the successor states, and then chose the successor which minimized the cost function. This formulation is only appropriate for discrete spaces, but fortunately, there is a continuous analog called **gradient descent**. The idea behind gradient descent is to take a series of small steps in the direction of steepest descent. This direction is given by the negative **gradient** of $J(\theta)$, where each component is the corresponding partial derivative of J.

In other words, we start with an arbitrary "initial guess" for θ, and then apply the following update rules:[3]

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta).$$

[3]We use $:=$ to denote assignment and $=$ to denote mathematical equality. For instance, $a := b$ is a computer operation assigning a variable a the value b, and $a = b$ is making the statement that a and b have the same value.

(This update is simultaneously performed for all values of $j = 0, \ldots, n$.) Here, α is called the **learning rate**.[4]

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. We have:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= \frac{1}{2} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= 2 \cdot \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x^{(i)}) - y^{(i)}) \\
&= \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{k=0}^{n} \theta_k x_k^{(i)} - y^{(i)} \right) \\
&= \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y\right) x_j^{(i)}
\end{aligned}$$

This gives the update rule:

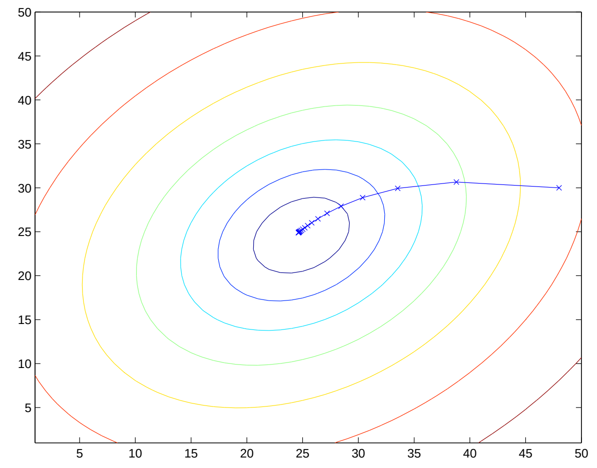
$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}.$$

The rule is called the **LMS** update rule (LMS stands for "least mean squares"). This rule has several properties that seem natural and intuitive. For instance, the magnitude of the update is proportional to the error term $(y^{(i)} - h_\theta(x^{(i)}))$; thus, for instance, if we are encountering a training example on which our prediction nearly matches the actual value of $y^{(i)}$, then we find that there is little need to change the parameters; in contrast, a larger change to the parameters will be made if our prediction $h_\theta(x^{(i)})$ has a large error (i.e., if it is very far from $y^{(i)}$).

The update rule we just described looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges

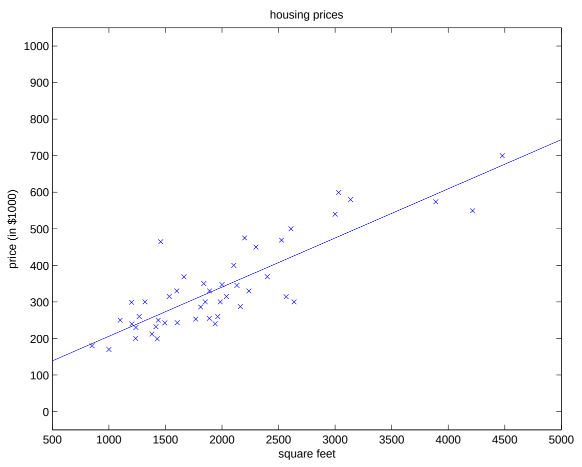
[4]The learning rate is a parameter of the algorithm which must be set by hand. Choosing the wrong value can lead to poor performance. Specifically, large values can cause gradient descent not to converge, while small values can cause it to converge too slowly.

(assuming the learning rate α is not too large) to the global minimum.[5] Here is an example of gradient descent as it is run to minimize a quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, with was initialized at (48,30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through.

When we run batch gradient descent to fit θ on our previous dataset, to learn to predict housing price as a function of living area, we obtain $\theta_0 = 71.27$, $\theta_1 = 0.1345$. If we plot $h_\theta(x)$ as a function of x (area), along with the training data, we obtain the following figure:



[5]More formally, it turns out that our least-squares cost function is **convex**, which implies it has no local optima.

If the number of bedrooms were included as one of the input features as well, we get $\theta_0 = 89.60, \theta_1 = 0.1392$, $\theta_2 = -8.738$.

The above results were obtained with batch gradient descent. There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

```
Loop {
    for i=1 to m, {
        $\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$    (for every $j$).
    }
}
```

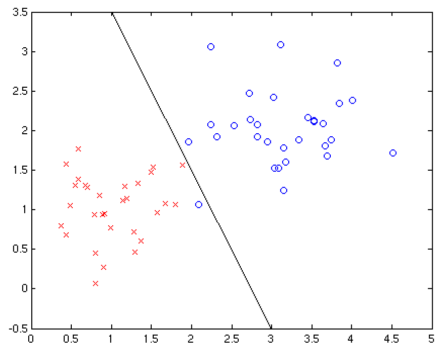
(We arrive at the update rule by taking the partial derivatives of $\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ with respect to θ_j.) In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called **stochastic gradient descent** (also **incremental gradient descent**). Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if m is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent. (Note however that it may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.[6]) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

3 Classification and logistic regression

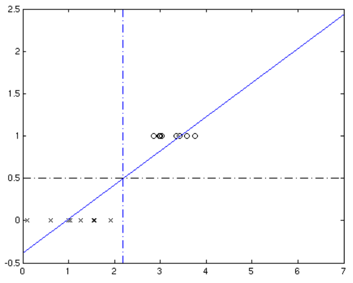
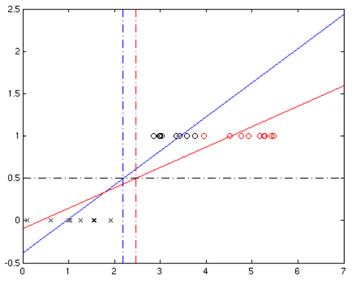
Lets now talk about classification. This is just like the regression problem, except that the values y we want to predict now take on only a small number of discrete values. For now, we will focus on the **binary classification**

[6]While it is more common to run stochastic gradient descent as we have described it and with a fixed learning rate α, by slowly letting the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather then merely oscillate around the minimum.

problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. 0 is also called the **negative class**, and 1 the **positive class**, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example. As before, we will focus on linear models, so our goal is to find a linear function which can distinguish one class from the other, such as the one shown below.



We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Furthermore, linear regression would lead to the counter-intuitive notion of predictions less than 0 or greater than 1. Linear regression also leads to the strange effect shown below. This figure represents a decision problem with a single feature, where we use linear regression to predict the 0 or 1 value of y given that one feature. The solid blue line shows the linear regression fit, and the dashed blue line shows where the decision boundary would lie if we use the arbitrary cutoff of 0.5. On the right, we add data points far to the right of the decision boundary. Counter-intuitively, this causes the estimated decision boundary to shift to the right.

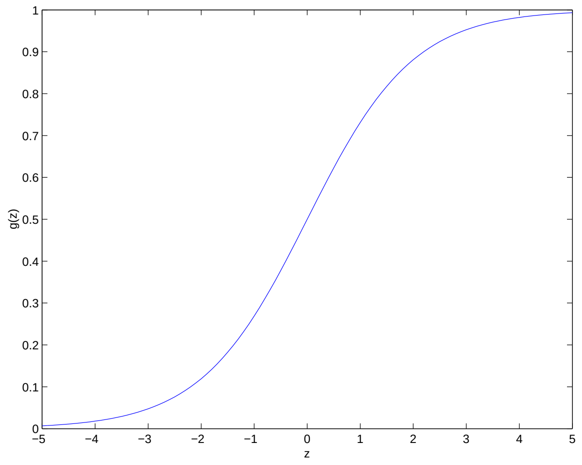
To fix this, lets change the form for our hypotheses $h_\theta(x)$. We will choose

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

where

$$g(z) = \frac{1}{1 + e^{-z}}$$

is called the **logistic function** or the **sigmoid function**. Here is a plot showing $g(z)$:



Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^n \theta_j x_j$.

Our objective function (which we are trying to maximize, rather than minimize) is the following:

$$\ell(\theta) = \sum_{i=1}^m y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))$$

As we show in the next section, this objective function can be maximized using the same batch gradient descent rule as before:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}.$$

4 Maximum likelihood

So far, we have presented recipes for the cost functions in linear regression and logistic regression. Now we show the justification behind these formulas. These two algorithms are both special cases of a general parameter estimation method called **maximum likelihood**. The intuition behind maximum likelihood is that we want to choose the hypothesis which makes the data as probable as possible. What do we mean by probable? To answer this question, we will have to define probabilistic models of how our data are generated.

As a motivating example, suppose we are given a (possibly biased) coin, and we want to determine the probability that this coin will come up heads. A coin toss can be modeled as a Bernoulli random variable, i.e. one which takes the value 1 with probability ϕ and 0 with probability $1 - \phi$. Suppose now that we toss this coin m times. This will give us m **independently and identically distributed (IID)** samples from this Bernoulli random variable. By IID, we mean that each sample is drawn from the same distribution, and each sample is independent of all of the others.

Let h represent the number of heads in the m tosses. You can check that the probability of seeing a particular sequence of exactly h heads is $m - h$ tails is:

$$\phi^h (1 - \phi)^{m-h}.$$

We call this probability the **likelihood**; it is the probability of the observed data assuming the model parameters. We often denote this probability as $p(\text{data}; \phi)$.[7]

In maximum likelihood, we choose our parameters to maximize the likelihood. Here, this gives us

$$\phi = \arg\max_\phi \; P(\text{data}; \phi) = \arg\max_\phi \phi^h (1 - \phi)^{m-h}.$$

Sometimes we can compute the maximum analytically by setting the partial derivatives equal to zero, and when this is not possible, we use an iterative

[7]Note that, in this class, we do *not* write $p(\text{data}|\phi)$, because we are not treating our parameter ϕ as a random variable, and therefore it makes no sense to condition on it.

procedure such as gradient descent. However, we don't usually apply either of these techniques to the likelihood itself, since it's awkward to take derivatives of products. Rather, we usually maximize the **log likelihood** $\log p(\text{data}; \phi)$, and this will always give us the same answer because the logarithm is a monotonically increasing function. In the case of coin flips,

$$\phi = \arg\max_{\phi} \log p(\text{data}; \phi) = \arg\max_{\phi} \; h \log \phi + (m-h) \log(1-\phi).$$

We can find the maximum likelihood solution by setting $\frac{\partial \log p(\text{data};\phi)}{\partial \phi}$ to zero:

$$\begin{aligned} \frac{\partial}{\partial \phi} \log p(\text{data}; \phi) &= \frac{\partial}{\partial \phi} \left(h \log \phi + (m-h) \log(1-\phi) \right) \\ &= \frac{h}{\phi} - \frac{m-h}{1-\phi} \\ \frac{\phi}{1-\pi} &= \frac{h}{m-h} \\ \phi &= \frac{h}{m}. \end{aligned}$$

Hence, the maximum likelihood estimate of ϕ turns out to be what we would expect: $\phi = \frac{h}{m}$, the proportion of flips which came up heads.

Before we move on, it's worth noting what maximum likelihood does *not* give us. It does not give us the probability of a given value of ϕ being the correct parameter. For instance, if you flip the coin once and it comes up heads, the likelihood for $\phi = 1$ would be 1, and the likelihood for $\phi = 0$ would be 0. But this doesn't mean we know that $\phi = 1$; in fact, we cannot make any statements about the probability of the parameters. (There is a branch of statistics called Bayesian statistics which does make such statements, but it is beyond the scope of this lecture.)

4.1 Linear regression

Now we'll show that linear regression and logistic regression are both special cases of maximum likelihood. For linear regression, we assume we have a fixed set of inputs $x^{(1)}, \ldots, x^{(m)}$. For each input $x^{(i)}$, we assume that $y^{(i)}$ is a Gaussian random variable whose mean is a linear function of $x^{(i)}$, and whose variance is fixed at σ^2. Mathematically,

$$p(y|x;\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - \theta^T x)^2}{2\sigma^2} \right).$$

Taking the log as we usually do,

$$\log p(y|x;\theta) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y-\theta^T x)^2}{2\sigma^2}.$$

When summed over all of our training examples,

$$\begin{aligned}
\log p(y^{(1)},\ldots,y^{(m)}|x^{(1)},\ldots,x^{(m)};\theta) &= \log\prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) \\
&= \sum_{i=1}^{m}\log p(y^{(i)}|x^{(i)};\theta) \\
&= -\frac{m}{2}\log(2\pi\sigma^2) - \sum_{i=1}^{m}\frac{(y-\theta^T x)^2}{2\sigma^2}
\end{aligned}$$

The $-\frac{m}{2}\log(2\pi\sigma^2)$ term doesn't depend on θ, and so maximizing this is equivalent to minimizing our least-squares objective function from before, so linear regression is a special case of maximum likelihood where the observations are assumed to be Gaussian. Note that we are treating the $x^{(i)}$'s as fixed, and the $y^{(i)}$'s as random variables.

4.2 Logistic regression

The derivation for logistic regression is similar. As in linear regression, we assume a fixed set of inputs $x^{(i)}$, and the targets $y^{(i)}$'s are treated as random variables. This time, the $y^{(i)}$'s are Bernoulli random variables whose parameter ϕ depends on x. In particular, we model ϕ as a logistic function of $\theta^T x^{(i)}$. Mathematically, we have

$$\begin{aligned}
p(y=1|x,\theta) &= h_\theta(x) \\
&= g(\theta^T x) \\
&= \frac{1}{1+\exp(-\theta^T x)} \\
\log p(y|x,\theta) &= y\log h_\theta(x) + (1-y)\log(1-h_\theta(x))
\end{aligned}$$

The log likelihood, therefore, is

$$\begin{aligned}
\log p(y^{(1)}, \ldots, y^{(m)} | x^{(1)}, \ldots, x^{(m)}; \theta) &= \log \prod_{i=1}^{m} p(y^{(i)} | x^{(i)}; \theta) \\
&= \sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}; \theta) \\
&= \sum_{i=1}^{m} y \log h_\theta(x) + (1-y) \log(1 - h_\theta(x))
\end{aligned}$$

We have arrived at the objective function we presented earlier for logistic regression.

Now we take the partial derivatives of the likelihood with respect to θ in order to derive the gradient descent update rules. Before moving on, here's a useful property of the derivative of the sigmoid function $g(z)$, which we write a g':

$$\begin{aligned}
g'(z) &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2} \left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1-g(z)).
\end{aligned}$$

Written in vectorial notation, our updates will be given by $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$. (Note the positive rather than negative sign in the update formula, since we're maximizing, rather than minimizing, a function now.) Lets start by working with just one training example (x, y), and take derivatives to derive the stochastic gradient ascent rule:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} \ell(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1 - g(\theta^T x)}\right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\
&= \left(y \frac{1}{g(\theta^T x)} - (1-y) \frac{1}{1 - g(\theta^T x)}\right) g(\theta^T x)(1 - g(\theta^T x) \frac{\partial}{\partial \theta_j} \theta^T x \\
&= \left(y(1 - g(\theta^T x)) - (1-y) g(\theta^T x)\right) x_j \\
&= \left(y - h_\theta(x)\right) x_j
\end{aligned}$$

Above, we used the fact that $g'(z) = g(z)(1-g(z))$. This therefore gives us the stochastic gradient ascent rule

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$$

The batch gradient descent rule will simply use the partial derivative summed over all of the training examples:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$$

Thus, logistic regression is simply the special case of maximum likelihood where the target variables are Bernoulli(ϕ), and ϕ is a sigmoidal function of x. As it turns out, maximum likelihood is a widely applicable formalism, and we will see one more example when we discuss decision tree learning in the next lecture.

Support vector machines

7

7.1 CS229 Lecture notes by Andrew Ng, Part V

CS229 Lecture notes

Andrew Ng

Part V

Support Vector Machines

This set of notes presents the Support Vector Machine (SVM) learning algorithm. SVMs are among the best (and many believe is indeed the best) "off-the-shelf" supervised learning algorithm. To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap." Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVMs efficiently in very high dimensional (such as infinite-dimensional) feature spaces, and finally, we'll close off the story with the SMO algorithm, which gives an efficient implementation of SVMs.

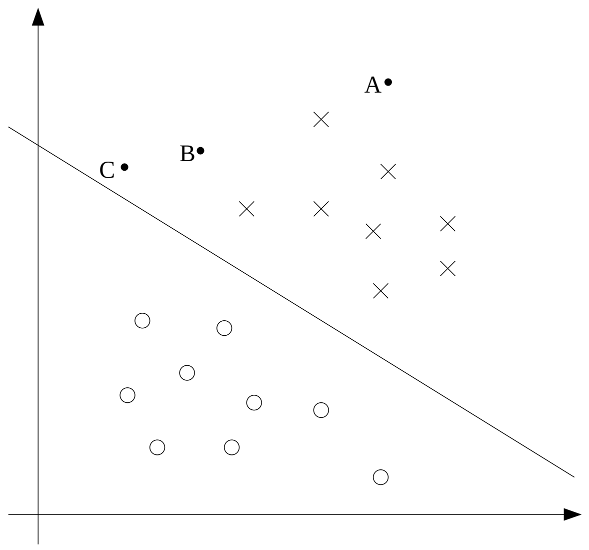
1 Margins: Intuition

We'll start our story on SVMs by talking about margins. This section will give the intuitions about margins and about the "confidence" of our predictions; these ideas will be made formal in Section 3.

Consider logistic regression, where the probability $p(y = 1|x; \theta)$ is modeled by $h_\theta(x) = g(\theta^T x)$. We would then predict "1" on an input x if and only if $h_\theta(x) \geq 0.5$, or equivalently, if and only if $\theta^T x \geq 0$. Consider a positive training example ($y = 1$). The larger $\theta^T x$ is, the larger also is $h_\theta(x) = p(y = 1|x; w, b)$, and thus also the higher our degree of "confidence" that the label is 1. Thus, informally we can think of our prediction as being a very confident one that $y = 1$ if $\theta^T x \gg 0$. Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\theta^T x \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^T x^{(i)} \gg 0$ whenever $y^{(i)} = 1$, and

$\theta^T x^{(i)} \ll 0$ whenever $y^{(i)} = 0$, since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which x's represent positive training examples, o's denote negative training examples, a decision boundary (this is the line given by the equation $\theta^T x = 0$, and is also called the **separating hyperplane**) is also shown, and three points have also been labeled A, B and C.



Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of y at at A, it seems we should be quite confident that $y = 1$ there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it'd be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

2 Notation

To make our discussion of SVMs easier, we'll first need to introduce a new notation for talking about classification. We will be considering a linear classifier for a binary classification problem with labels y and features x. From now, we'll use $y \in \{-1, 1\}$ (instead of $\{0, 1\}$) to denote the class labels. Also, rather than parameterizing our linear classifier with the vector θ, we will use parameters w, b, and write our classifier as

$$h_{w,b}(x) = g(w^T x + b).$$

Here, $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise. This "w, b" notation allows us to explicitly treat the intercept term b separately from the other parameters. (We also drop the convention we had previously of letting $x_0 = 1$ be an extra coordinate in the input feature vector.) Thus, b takes the role of what was previously θ_0, and w takes the role of $[\theta_1 \dots \theta_n]^T$.

Note also that, from our definition of g above, our classifier will directly predict either 1 or -1 (cf. the perceptron algorithm), without first going through the intermediate step of estimating the probability of y being 1 (which was what logistic regression did).

3 Functional and geometric margins

Lets formalize the notions of the functional and geometric margins. Given a training example $(x^{(i)}, y^{(i)})$, we define the **functional margin** of (w, b) with respect to the training example

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b).$$

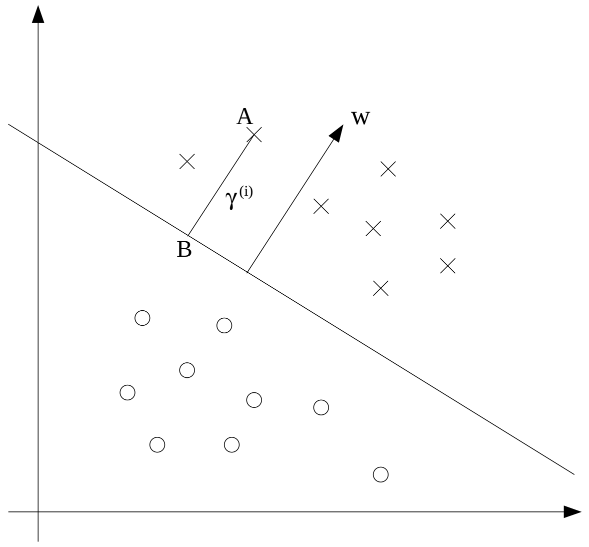
Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), then we need $w^T x + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, then we need $w^T x + b$ to be a large negative number. Moreover, if $y^{(i)}(w^T x + b) > 0$, then our prediction on this example is correct. (Check this yourself.) Hence, a large functional margin represents a confident and a correct prediction.

For a linear classifier with the choice of g given above (taking values in $\{-1, 1\}$), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace w with $2w$ and b with $2b$, then since $g(w^T x + b) = g(2w^T x + 2b)$, this would not change $h_{w,b}(x)$ at all. I.e., g, and hence also $h_{w,b}(x)$, depends only on the sign, but not on the magnitude, of $w^T x + b$. However, replacing (w, b) with $(2w, 2b)$ also results in multiplying our functional margin by a factor of 2. Thus, it seems that by exploiting our freedom to scale w and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition such as that $||w||_2 = 1$; i.e., we might replace (w, b) with $(w/||w||_2, b/||w||_2)$, and instead consider the functional margin of $(w/||w||_2, b/||w||_2)$. We'll come back to this later.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the function margin of (w, b) with respect to S as the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}.$$

Next, lets talk about **geometric margins**. Consider the picture below:



The decision boundary corresponding to (w, b) is shown, along with the vector w. Note that w is orthogonal (at 90°) to the separating hyperplane. (You should convince yourself that this must be the case.) Consider the point at A, which represents the input $x^{(i)}$ of some training example with label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$, is given by the line segment AB.

How can we find the value of $\gamma^{(i)}$? Well, $w/||w||$ is a unit-length vector pointing in the same direction as w. Since A represents $x^{(i)}$, we therefore find that the point B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. But this point lies on the decision boundary, and all points x on the decision boundary satisfy the equation $w^T x + b = 0$. Hence,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0.$$

Solving for $\gamma^{(i)}$ yields

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}.$$

This was worked out for the case of a positive training example at A in the figure, where being on the "positive" side of the decision boundary is good. More generally, we define the geometric margin of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right).$$

Note that if $||w|| = 1$, then the functional margin equals the geometric margin—this thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters; i.e., if we replace w with $2w$ and b with $2b$, then the geometric margin does not change. This will in fact come in handy later. Specifically, because of this invariance to the scaling of the parameters, when trying to fit w and b to training data, we can impose an arbitrary scaling constraint on w without changing anything important; for instance, we can demand that $||w|| = 1$, or $|w_1| = 5$, or $|w_1 + b| + |w_2| = 2$, and any of these can be satisfied simply by rescaling w and b.

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}.$$

4 The optimal margin classifier

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions on the training set and a good "fit" to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a "gap" (geometric margin).

For now, we will assume that we are given a training set that is linearly separable; i.e., that it is possible to separate the positive and negative examples using some separating hyperplane. How we we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, \ldots, m \\ & ||w|| = 1. \end{aligned}$$

I.e., we want to maximize γ, subject to each training example having functional margin at least γ. The $||w|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ. Thus, solving this problem will result in (w, b) with the largest possible geometric margin with respect to the training set.

If we could solve the optimization problem above, we'd be done. But the "$||w|| = 1$" constraint is a nasty (non-convex) one, and this problem certainly isn't in any format that we can plug into standard optimization software to solve. So, lets try transforming the problem into a nicer one. Consider:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \frac{\hat{\gamma}}{||w||} \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, \ldots, m \end{aligned}$$

Here, we're going to maximize $\hat{\gamma}/||w||$, subject to the functional margins all being at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \hat{\gamma}/||w|$, this will give us the answer we want. Moreover, we've gotten rid of the constraint $||w|| = 1$ that we didn't like. The downside is that we now have a nasty (again, non-convex) objective $\frac{\hat{\gamma}}{||w||}$ function; and, we still don't have any off-the-shelf software that can solve this form of an optimization problem.

Lets keep going. Recall our earlier discussion that we can add an arbitrary scaling constraint on w and b without changing anything. This is the key idea we'll use now. We will introduce the scaling constraint that the functional margin of w, b with respect to the training set must be 1:

$$\hat{\gamma} = 1.$$

Since multiplying w and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling w, b. Plugging this into our problem above, and noting that maximizing $\hat{\gamma}/||w|| = 1/||w||$ is the same thing as minimizing $||w||^2$, we now have the following optimization problem:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m \end{aligned}$$

We've now transformed the problem into a form that can be efficiently solved. The above is an optimization problem with a convex quadratic objective and only linear constraints. Its solution gives us the **optimal margin classifier**. This optimization problem can be solved using commercial quadratic programming (QP) code.[1]

While we could call the problem solved here, what we will instead do is make a digression to talk about Lagrange duality. This will lead us to our optimization problem's dual form, which will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

5 Lagrange duality

Lets temporarily put aside SVMs and maximum margin classifiers, and talk about solving constrained optimization problems.

Consider a problem of the following form:

$$\begin{aligned} \min_{w} \quad & f(w) \\ \text{s.t.} \quad & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

Some of you may recall how the method of Lagrange multipliers can be used to solve it. (Don't worry if you haven't seen it before.) In this method, we define the **Lagrangian** to be

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

[1]You may be familiar with linear programming, which solves optimization problems that have linear objectives and linear constraints. QP software is also widely available, which allows convex quadratic objectives and linear constraints.

Here, the β_i's are called the **Lagrange multipliers**. We would then find and set $\mathcal{L}$'s partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial w_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0,$$

and solve for w and β.

In this section, we will generalize this to constrained optimization problems in which we may have inequality as well as equality constraints. Due to time constraints, we won't really be able to do the theory of Lagrange duality justice in this class,[2] but we will give the main ideas and results, which we will then apply to our optimal margin classifier's optimization problem.

Consider the following, which we'll call the **primal** optimization problem:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{s.t.} \quad & g_i(w) \le 0, \quad i = 1, \ldots, k \\ & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

To solve it, we start by defining the **generalized Lagrangian**

$$\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w).$$

Here, the α_i's and β_i's are the Lagrange multipliers. Consider the quantity

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{P}$" subscript stands for "primal." Let some w be given. If w violates any of the primal constraints (i.e., if either $g_i(w) > 0$ or $h_i(w) \neq 0$ for some i), then you should be able to verify that

$$\begin{aligned} \theta_{\mathcal{P}}(w) &= \max_{\alpha, \beta \,:\, \alpha_i \ge 0} f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w) \qquad (1) \\ &= \infty. \qquad (2) \end{aligned}$$

Conversely, if the constraints are indeed satisfied for a particular value of w, then $\theta_{\mathcal{P}}(w) = f(w)$. Hence,

$$\theta_{\mathcal{P}}(w) = \begin{cases} f(w) & \text{if } w \text{ satisfies primal constraints} \\ \infty & \text{otherwise.} \end{cases}$$

[2]Readers interested in learning more about this topic are encouraged to read, e.g., R. T. Rockarfeller (1970), Convex Analysis, Princeton University Press.

Thus, $\theta_{\mathcal{P}}$ takes the same value as the objective in our problem for all values of w that satisfies the primal constraints, and is positive infinity if the constraints are violated. Hence, if we consider the minimization problem

$$\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha,\beta\,:\,\alpha_i\geq 0} \mathcal{L}(w,\alpha,\beta),$$

we see that it is the same problem (i.e., and has the same solutions as) our original, primal problem. For later use, we also define the optimal value of the objective to be $p^* = \min_w \theta_{\mathcal{P}}(w)$; we call this the **value** of the primal problem.

Now, lets look at a slightly different problem. We define

$$\theta_{\mathcal{D}}(\alpha,\beta) = \min_w \mathcal{L}(w,\alpha,\beta).$$

Here, the "$\mathcal{D}$" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α,β, here are are minimizing with respect to w.

We can now pose the **dual** optimization problem:

$$\max_{\alpha,\beta\,:\,\alpha_i\geq 0} \theta_{\mathcal{D}}(\alpha,\beta) = \max_{\alpha,\beta\,:\,\alpha_i\geq 0} \min_w \mathcal{L}(w,\alpha,\beta).$$

This is exactly the same as our primal problem shown above, except that the order of the "max" and the "min" are now exchanged. We also define the optimal value of the dual problem's objective to be $d^* = \max_{\alpha,\beta\,:\,\alpha_i\geq 0} \theta_{\mathcal{D}}(w)$.

How are the primal and the dual problems related? It can easily be shown that

$$d^* = \max_{\alpha,\beta\,:\,\alpha_i\geq 0} \min_w \mathcal{L}(w,\alpha,\beta) \leq \min_w \max_{\alpha,\beta\,:\,\alpha_i\geq 0} \mathcal{L}(w,\alpha,\beta) = p^*.$$

(You should convince yourself of this; this follows from the "max min" of a function always being less than or equal to the "min max.") However, under certain conditions, we will have

$$d^* = p^*,$$

so that we can solve the dual problem in lieu of the primal problem. Lets see what these conditions are.

Suppose f and the g_i's are convex,[3] and the h_i's are affine.[4] Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i.

[3]When f has a Hessian, then it is convex if and only if the hessian is positive semi-definite. For instance, $f(w) = w^T w$ is convex; similarly, all linear (and affine) functions are also convex. (A function f can also be convex without being differentiable, but we won't need those more general definitions of convexity here.)

[4]I.e., there exists a_i, b_i, so that $h_i(w) = a_i^T w + b_i$. "Affine" means the same thing as linear, except that we also allow the extra intercept term b_i.

Under our above assumptions, there must exist w^*, α^*, β^* so that w^* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and moreover $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$. Moreover, w^*, α^* and β^* satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\begin{aligned}
\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, n & (3) \\
\frac{\partial}{\partial \beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, l & (4) \\
\alpha_i^* g_i(w^*) &= 0, \quad i = 1, \ldots, k & (5) \\
g_i(w^*) &\leq 0, \quad i = 1, \ldots, k & (6) \\
\alpha^* &\geq 0, \quad i = 1, \ldots, k & (7)
\end{aligned}$$

Moreover, if some w^*, α^*, β^* satisfy the KKT conditions, then it is also a solution to the primal and dual problems.

We draw attention to Equation (5), which is called the KKT **dual complementarity** condition. Specifically, it implies that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. (I.e., the "$g_i(w) \leq 0$" constraint is **active**, meaning it holds with equality rather than with inequality.) Later on, this will be key for showing that the SVM has only a small number of "support vectors"; the KKT dual complementarity condition will also give us our convergence test when we talk about the SMO algorithm.

6 Optimal margin classifiers

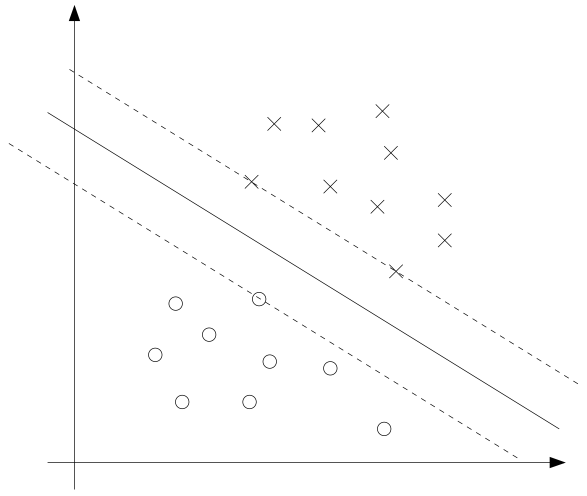
Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

$$\begin{aligned}
\min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\
\text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m
\end{aligned}$$

We can write the constraints as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



The points with the smallest margins are exactly the ones closest to the decision boundary; here, these are the three points (one negative and two positive examples) that lie on the dashed lines parallel to the decision boundary. Thus, only three of the α_i's—namely, the ones corresponding to these three training examples—will be non-zero at the optimal solution to our optimization problem. These three points are called the **support vectors** in this problem. The fact that the number of support vectors can be much smaller than the size the training set will be useful later.

Lets move on. Looking ahead, as we develop the dual form of the problem, one key idea to watch out for is that we'll try to write our algorithm in terms of only the inner product $\langle x^{(i)}, x^{(j)} \rangle$ (think of this as $(x^{(i)})^T x^{(j)}$) between points in the input feature space. The fact that we can express our algorithm in terms of these inner products will be key when we apply the kernel trick.

When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]. \tag{8}$$

Note that there're only "α_i" but no "β_i" Lagrange multipliers, since the problem has only inequality constraints.

Lets find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b (for fixed α), to get $\theta_{\mathcal{D}}$, which we'll do by setting the derivatives of $\mathcal{L}$ with respect to w and b to zero. We have:

$$\nabla_w \mathcal{L}(w,b,\alpha) = w - \sum_{i=1}^m \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^m \alpha_i y^{(i)} x^{(i)}. \tag{9}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b}\mathcal{L}(w,b,\alpha) = \sum_{i=1}^m \alpha_i y^{(i)} = 0. \tag{10}$$

If we take the definition of w in Equation (9) and plug that back into the Lagrangian (Equation 8), and simplify, we get

$$\mathcal{L}(w,b,\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j (x^{(i)})^T x^{(j)} - b\sum_{i=1}^m \alpha_i y^{(i)}.$$

But from Equation (10), the last term must be zero, so we obtain

$$\mathcal{L}(w,b,\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j (x^{(i)})^T x^{(j)}.$$

Recall that we got to the equation above by minimizing $\mathcal{L}$ with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ (that we always had) and the constraint (10), we obtain the following dual optimization problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j \langle x^{(i)}, x^{(j)}\rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, m \\ & \sum_{i=1}^m \alpha_i y^{(i)} = 0, \end{aligned}$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations 3–7) to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's. We'll talk later about the specific algorithm that we're going to use to solve the dual problem, but if we are indeed able to solve it (i.e., find the α's that maximize $W(\alpha)$ subject to the constraints), then we can use Equation (9) to go back and find the optimal w's as a function of the α's. Having found w^*, by considering the primal problem, it is also straightforward to find the optimal value for the intercept term b as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T}x^{(i)} + \min_{i:y^{(i)}=1} w^{*T}x^{(i)}}{2}. \tag{11}$$

(Check for yourself that this is correct.)

Before moving on, lets also take a more careful look at Equation (9), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^Tx + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (9), this quantity can also be written:

$$\begin{aligned} w^Tx + b &= \left(\sum_{i=1}^m \alpha_i y^{(i)} x^{(i)}\right)^T x + b & (12) \\ &= \sum_{i=1}^m \alpha_i y^{(i)} \langle x^{(i)}, x\rangle + b. & (13) \end{aligned}$$

Hence, if we've found the α_i's, in order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate (13) and make our prediction.

By examining the dual form of the optimization problem, we gained significant insight into the structure of the problem, and were also able to write the entire algorithm in terms of only inner products between input feature vectors. In the next section, we will exploit this property to apply the kernels to our classification problem. The resulting algorithm, **support vector machines**, will be able to efficiently learn in very high dimensional spaces.

7 Kernels

Back in our discussion of linear regression, we had a problem in which the input x was the living area of a house, and we considered performing regression using the features x, x^2 and x^3 (say) to obtain a cubic function. To distinguish between these two sets of variables, we'll call the "original" input value the input **attributes** of a problem (in this case, x, the living area). When that is mapped to some new set of quantities that are then passed to the learning algorithm, we'll call those new quantities the input **features**. (Unfortunately, different authors use different terms to describe these two things, but we'll try to use this terminology consistently in these notes.) We will also let ϕ denote the **feature mapping**, which maps from the attributes to the features. For instance, in our example, we had

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}.$$

Rather than applying SVMs using the original input attributes x, we may instead want to learn using some features $\phi(x)$. To do so, we simply need to go over our previous algorithm, and replace x everywhere in it with $\phi(x)$.

Since the algorithm can be written entirely in terms of the inner products $\langle x, z\rangle$, this means that we would replace all those inner products with $\langle \phi(x), \phi(z)\rangle$. Specificically, given a feature mapping ϕ, we define the corresponding **Kernel** to be

$$K(x, z) = \phi(x)^T \phi(z).$$

Then, everywhere we previously had $\langle x, z\rangle$ in our algorithm, we could simply replace it with $K(x, z)$, and our algorithm would now be learning using the features ϕ.

Now, given ϕ, we could easily compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. But what's more interesting is that often, $K(x, z)$ may be very inexpensive to calculate, even though $\phi(x)$ itself may be very expensive to calculate (perhaps because it is an extremely high dimensional vector). In such settings, by using in our algorithm an efficient way to calculate $K(x, z)$, we can get SVMs to learn in the high dimensional feature space space given by ϕ, but without ever having to explicitly find or represent vectors $\phi(x)$.

Lets see an example. Suppose $x, z \in \mathbb{R}^n$, and consider

$$K(x, z) = (x^T z)^2.$$

We can also write this as

$$\begin{aligned} K(x,z) &= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_i z_i\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) \end{aligned}$$

Thus, we see that $K(x,z) = \phi(x)^T\phi(z)$, where the feature mapping ϕ is given (shown here for the case of $n = 3$) by

$$\phi(x) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \end{bmatrix}.$$

Note that whereas calculating the high-dimensional $\phi(x)$ requires $O(n^2)$ time, finding $K(x,z)$ takes only $O(n)$ time—linear in the dimension of the input attributes.

For a related kernel, also consider

$$\begin{aligned} K(x,z) &= (x^Tz + c)^2 \\ &= \sum_{i,j=1}^{n}(x_ix_j)(z_iz_j) + \sum_{i=1}^{n}(\sqrt{2c}x_i)(\sqrt{2c}z_i) + c^2. \end{aligned}$$

(Check this yourself.) This corresponds to the feature mapping (again shown for $n = 3$)

$$\phi(x) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \sqrt{2c}x_3 \\ c \end{bmatrix},$$

and the parameter c controls the relative weighting between the x_i (first order) and the x_ix_j (second order) terms.

More broadly, the kernel $K(x, z) = (x^Tz + c)^d$ corresponds to a feature mapping to an $\binom{n+d}{d}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\dots x_{i_k}$ that are up to order d. However, despite working in this $O(n^d)$-dimensional space, computing $K(x, z)$ still takes only $O(n)$ time, and hence we never need to explicitly represent feature vectors in this very high dimensional feature space.

Now, lets talk about a slightly different view of kernels. Intuitively, (and there are things wrong with this intuition, but nevermind), if $\phi(x)$ and $\phi(z)$ are close together, then we might expect $K(x, z) = \phi(x)^T\phi(z)$ to be large. Conversely, if $\phi(x)$ and $\phi(z)$ are far apart—say nearly orthogonal to each other—then $K(x, z) = \phi(x)^T\phi(z)$ will be small. So, we can think of $K(x, z)$ as some measurement of how similar are $\phi(x)$ and $\phi(z)$, or of how similar are x and z.

Given this intuition, suppose that for some learning problem that you're working on, you've come up with some function $K(x, z)$ that you think might be a reasonable measure of how similar x and z are. For instance, perhaps you chose

$$K(x, z) = \exp\left(-\frac{||x - z||^2}{2\sigma^2}\right).$$

This is a resonable measure of x and z's similarity, and is close to 1 when x and z are close, and near 0 when x and z are far apart. Can we use this definition of K as the kernel in an SVM? In this particular example, the answer is yes. (This kernel is called the **Gaussian kernel**, and corresponds to an infinite dimensional feature mapping ϕ.) But more broadly, given some function K, how can we tell if it's a valid kernel; i.e., can we tell if there is some feature mapping ϕ so that $K(x,z) = \phi(x)^T\phi(z)$ for all x, z?

Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ. Now, consider some finite set of m points (not necessarily the training set) $\{x^{(1)}, \dots, x^{(m)}\}$, and let a square, m-by-m matrix K be defined so that its (i,j)-entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the **Kernel matrix**. Note that we've overloaded the notation and used K to denote both the kernel function $K(x,z)$ and the kernel matrix K, due to their obvious close relationship.

Now, if K is a valid Kernel, then $K_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T\phi(x^{(j)}) = \phi(x^{(j)})^T\phi(x^{(i)}) = K(x^{(j)}, x^{(i)}) = K_{ji}$, and hence K must be symmetric. Moreover, letting $\phi_k(x)$ denote the k-th coordinate of the vector $\phi(x)$, we find that for any vector z, we have

$$\begin{aligned}
z^TKz &= \sum_i\sum_j z_iK_{ij}z_j \\
&= \sum_i\sum_j z_i\phi(x^{(i)})^T\phi(x^{(j)})z_j \\
&= \sum_i\sum_j z_i\sum_k \phi_k(x^{(i)})\phi_k(x^{(j)})z_j \\
&= \sum_k\sum_i\sum_j z_i\phi_k(x^{(i)})\phi_k(x^{(j)})z_j \\
&= \sum_k\left(\sum_i z_i\phi_k(x^{(i)})\right)^2 \\
&\geq 0.
\end{aligned}$$

The second-to-last step above used the same trick as you saw in Problem set 1 Q1. Since z was arbitrary, this shows that K is positive semi-definite ($K \geq 0$).

Hence, we've shown that if K is a valid kernel (i.e., if it corresponds to some feature mapping ϕ), then the corresponding Kernel matrix $K \in \mathbb{R}^{m\times m}$ is symmetric positive semidefinite. More generally, this turns out to be not only a necessary, but also a sufficient, condition for K to be a valid kernel (also called a Mercer kernel). The following result is due to Mercer.[5]

[5] Many texts present Mercer's theorem in a slightly more complicated form involving L^2 functions, but when the input attributes take values in $\mathbb{R}^n$, the version given here is equivalent.

Theorem (Mercer). Let $K : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, \ldots, x^{(m)}\}$, $(m < \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

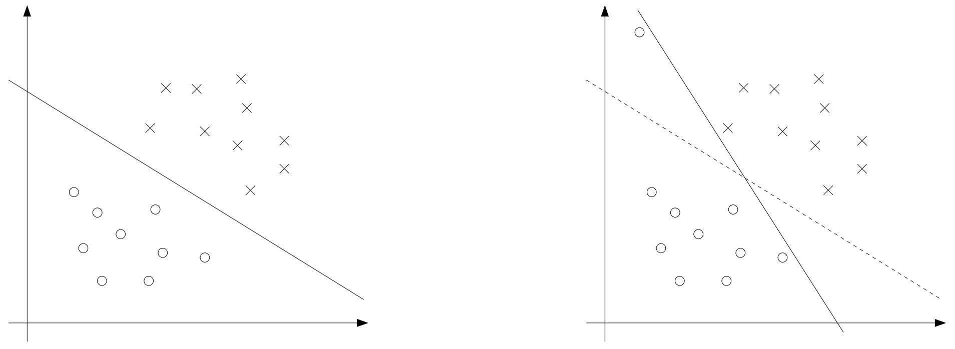
Given a function K, apart from trying to find a feature mapping ϕ that corresponds to it, this theorem therefore gives another way of testing if it is a valid kernel. You'll also have a chance to play with these ideas more in problem set 2.

In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16x16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^T z)^d$ or the Gaussian kernel, SVMs were able to obtain extremely good performance on this problem. This was particularly surprising since the input attributes x were just a 256-dimensional vector of the image pixel intensity values, and the system had no prior knowledge about vision, or even about which pixels are adjacent to which other ones. Another example that we briefly talked about in lecture was that if the objects x that we are trying to classify are strings (say, x is a list of amino acids, which strung together form a protein), then it seems hard to construct a reasonable, "small" set of features for most learning algorithms, especially if different strings have different lengths. However, consider letting $\phi(x)$ be a feature vector that counts the number of occurrences of each length-k substring in x. If we're considering strings of english alphabets, then there're 26^k such strings. Hence, $\phi(x)$ is a 26^k dimensional vector; even for moderate values of k, this is probably too big for us to efficiently work with. (e.g., $26^4 \approx 460000$.) However, using (dynamic programming-ish) string matching algorithms, it is possible to efficiently compute $K(x, z) = \phi(x)^T \phi(z)$, so that we can now implicitly work in this 26^k-dimensional feature space, but without ever explicitly computing feature vectors in this space.

The application of kernels to support vector machines should already be clear and so we won't dwell too much longer on it here. Keep in mind however that the idea of kernels has significantly broader applicability than SVMs. Specifically, if you have any learning algorithm that you can write in terms of only inner products $\langle x, z \rangle$ between input attribute vectors, then by replacing this with $K(x, z)$ where K is a kernel, you can "magically" allow your algorithm to work efficiently in the high dimensional feature space corresponding to K. For instance, this kernel trick can be applied with the perceptron to to derive a kernel perceptron algorithm. Many of the algorithms that we'll see later in this class will also be amenable to this method, which has come to be known as the "kernel trick."

8 Regularization and the non-separable case

The derivation of the SVM as presented so far assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers. For instance, the left figure below shows an optimal margin classifier, and when a single outlier is added in the upper-left region (right figure), it causes the decision boundary to make a dramatic swing, and the resulting classifier has a much smaller margin.



To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization (using ℓ_1 **regularization**) as follows:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, m \\ & \xi_i \geq 0, \quad i = 1, \ldots, m. \end{aligned}$$

Thus, examples are now permitted to have (functional) margin less than 1, and if an example whose functional margin is $1 - \xi_i$, we would pay a cost of the objective function being increased by $C\xi_i$. The parameter C controls the relative weighting between the twin goals of making the $||w||^2$ large (which we saw earlier makes the margin small) and of ensuring that most examples have functional margin at least 1.

As before, we can form the Lagrangian:

$$\mathcal{L}(w,b,\xi,\alpha,r) = \frac{1}{2}w^Tw + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \alpha_i\left[y^{(i)}(x^Tw+b)-1+\xi_i\right] - \sum_{i=1}^m r_i\xi_i.$$

Here, the α_i's and r_i's are our Lagrange multipliers (constrained to be ≥ 0). We won't go through the derivation of the dual again in detail, but after setting the derivatives with respect to w and b to zero as before, substituting them back in, and simplifying, we obtain the following dual form of the problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j\langle x^{(i)}, x^{(j)}\rangle \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1,\dots,m \\ & \sum_{i=1}^m \alpha_i y^{(i)} = 0, \end{aligned}$$

As before, we also have that w can be expressed in terms of the α_i's as given in Equation (9), so that after solving the dual problem, we can continue to use Equation (13) to make our predictions. Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (Equation 11 is no longer valid); see the comments in the next section/Platt's paper.

Also, the KKT dual-complementarity conditions (which in the next section will be useful for testing for the convergence of the SMO algorithm) are:

$$\alpha_i = 0 \;\Rightarrow\; y^{(i)}(w^Tx^{(i)}+b) \geq 1 \tag{14}$$

$$\alpha_i = C \;\Rightarrow\; y^{(i)}(w^Tx^{(i)}+b) \leq 1 \tag{15}$$

$$0 < \alpha_i < C \;\Rightarrow\; y^{(i)}(w^Tx^{(i)}+b) = 1. \tag{16}$$

Now, all that remains is to give an algorithm for actually solving the dual problem, which we will do in the next section.

9 The SMO algorithm

The SMO (sequential minimal optimization) algorithm, due to John Platt, gives an efficient way of solving the dual problem arising from the derivation of the SVM. Partly to motivate the SMO algorithm, and partly because it's interesting in its own right, lets first take another digression to talk about the coordinate ascent algorithm.

9.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

$$\max_{\alpha} W(\alpha_1, \alpha_2, \ldots, \alpha_m).$$

Here, we think of W as just some function of the parameters α_i's, and for now ignore any relationship between this problem and SVMs. We've already seen two optimization algorithms, gradient ascent and Newton's method. The new algorithm we're going to consider here is called **coordinate ascent**:

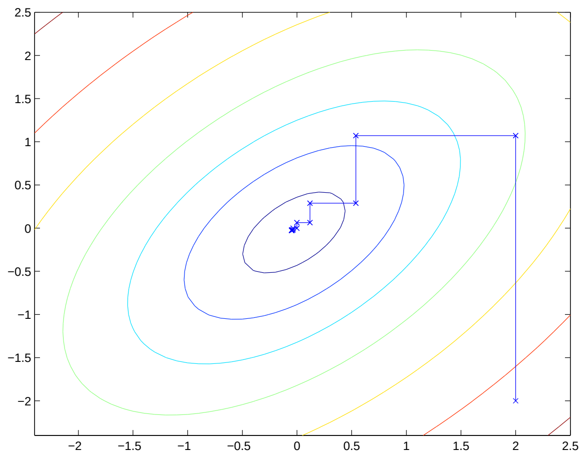
```
Loop until convergence: {

    For $i = 1, \ldots, m$, {
        $\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_m)$.
    }

}
```

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some α_i fixed, and reoptimize W with respect to just the parameter α_i. In the version of this method presented here, the inner-loop reoptimizes the variables in order $\alpha_1, \alpha_2, \ldots, \alpha_m, \alpha_1, \alpha_2, \ldots$. (A more sophisticated version might choose other orderings; for instance, we may choose the next variable to update according to which one we expect to allow us to make the largest increase in $W(\alpha)$.)

When the function W happens to be of such a form that the "arg max" in the inner loop can be performed efficiently, then coordinate ascent can be a fairly efficient algorithm. Here's a picture of coordinate ascent in action:



The ellipses in the figure are the contours of a quadratic function that we want to optimize. Coordinate ascent was initialized at $(2, -2)$, and also plotted in the figure is the path that it took on its way to the global maximum. Notice that on each step, coordinate ascent takes a step that's parallel to one of the axes, since only one variable is being optimized at a time.

9.2 SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm. Some details will be left to the homework, and for others you may refer to the paper excerpt handed out in class.

Here's the (dual) optimization problem that we want to solve:

$$\max_\alpha \quad W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{17}$$

$$\text{s.t.} \quad 0 \le \alpha_i \le C, \quad i = 1, \ldots, m \tag{18}$$

$$\sum_{i=1}^m \alpha_i y^{(i)} = 0. \tag{19}$$

Lets say we have set of α_i's that satisfy the constraints (18-19). Now, suppose we want to hold $\alpha_2, \ldots, \alpha_m$ fixed, and take a coordinate ascent step and reoptimize the objective with respect to α_1. Can we make any progress? The answer is no, because the constraint (19) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^m \alpha_i y^{(i)}.$$

Or, by multiplying both sides by $y^{(1)}$, we equivalently have

$$\alpha_1 = -y^{(1)} \sum_{i=2}^{m} \alpha_i y^{(i)}.$$

(This step used the fact that $y^{(1)} \in \{-1, 1\}$, and hence $(y^{(1)})^2 = 1$.) Hence, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, \ldots, \alpha_m$ fixed, then we can't make any change to α_1 without violating the constraint (19) in the optimization problem.

Thus, if we want to update some subject of the α_i's, we must update at least two of them simultaneously in order to keep satisfying the constraints. This motivates the SMO algorithm, which simply does the following:

```
Repeat till convergence {
    1. Select some pair $\alpha_i$ and $\alpha_j$ to update next (using a heuristic that
       tries to pick the two that will allow us to make the biggest progress
       towards the global maximum).
    2. Reoptimize $W(\alpha)$ with respect to $\alpha_i$ and $\alpha_j$, while holding all the
       other $\alpha_k$'s ($k \neq i, j$) fixed.
}
```

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 14-16) are satisfied to within some *tol*. Here, *tol* is the convergence tolerance parameter, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

The key reason that SMO is an efficient algorithm is that the update to α_i, α_j can be computed very efficiently. Lets now briefly sketch the main ideas for deriving the efficient update.

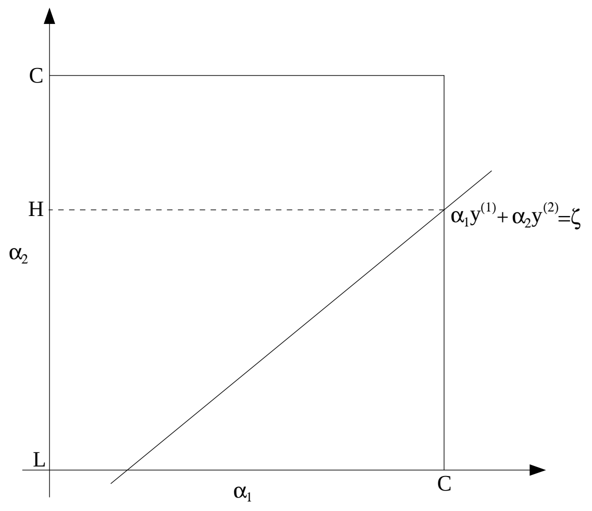
Lets say we currently have some setting of the α_i's that satisfy the constraints (18-19), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_m$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_m)$ with respect to α_1 and α_2 (subject to the constraints). From (19), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{m} \alpha_i y^{(i)}.$$

Since the right hand side is fixed (as we've fixed $\alpha_3, \ldots \alpha_m$), we can just let it be denoted by some constant ζ:

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \tag{20}$$

We can thus picture the constraints on α_1 and α_2 as follows:



From the constraints (18), we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Also plotted is the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$, on which we know α_1 and α_2 must lie. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$; otherwise, (α_1, α_2) can't simultaneously satisfy both the box and the straight line constraint. In this example, $L = 0$. But depending on what the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ looks like, this won't always necessarily be the case; but more generally, there will be some lower-bound L and some upper-bound H on the permissable values for α_2 that will ensure that α_1, α_2 lie within the box $[0, C] \times [0, C]$.

Using Equation (20), we can also write α_1 as a function of α_2:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}.$$

(Check this derivation yourself; we again used the fact that $y^{(1)} \in \{-1, 1\}$ so that $(y^{(1)})^2 = 1$.) Hence, the objective $W(\alpha)$ can be written

$$W(\alpha_1, \alpha_2, \ldots, \alpha_m) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, \ldots, \alpha_m).$$

Treating $\alpha_3, \ldots, \alpha_m$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints (18) (or, equivalently, that $L \leq \alpha_2 \leq H$), then we can easily maximize this quadratic function by setting its derivative to zero and solving. We'll let $\alpha_2^{new,unclipped}$ denote the resulting value of α_2. You should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{new,unclipped}$ and "clipping" it to lie in the

$[L, H]$ interval, to get

$$\alpha_2^{new} = \begin{cases} H & \text{if } \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & \text{if } L \leq \alpha_2^{new,unclipped} \leq H \\ L & \text{if } \alpha_2^{new,unclipped} < L \end{cases}$$

Finally, having found the α_2^{new}, we can use Equation (20) to go back and find the optimal value of α_1^{new}.

There're a couple more details that are quite easy but that we'll leave you to read about yourself in Platt's paper: One is the choice of the heuristics used to select the next α_i, α_j to update; the other is how to update b as the SMO algorithm is run.

8 Learning Theory

8.1 CS221 Lecture notes by Andrew Ng, No 7

see also CS229 Lecture notes by Andrew Ng, Part VI

CS221 Lecture notes #7

Supervised learning summary

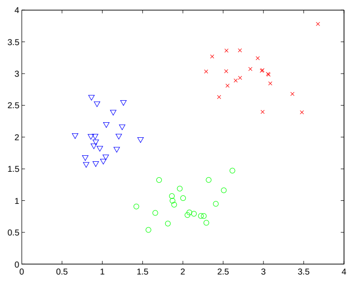
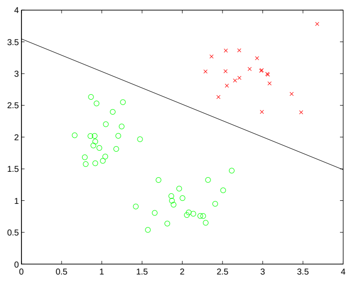
In the previous sets of notes on supervised learning, we discussed many specific algorithms for supervised learning. Now, we're going to take a step back and discuss some of the principles of how to use these learning algorithms to achieve good performance.

1 Multi-class classification

When discussing logistic regression and decision trees, we simplified our task by focusing on binary classification tasks, where there are only two categories to distinguish. However, many problems require us to distinguish more than two categories. Many binary classification algorithms can be extended to directly deal with multiple classes, but there is one general approach we can take even for algorithms which don't have straightforward multiclass extensions.

In **one-vs.-all** (also called **one-vs.-many** or **one-vs.-rest**), if we are trying to distinguish between N different classes, we train N different classifiers, each one of which tries to distinguish one class from all the rest. For instance, suppose we are given the three-class data shown in Figure 1 (a). We construct three different classification problems, each of which uses one of the three classes for the positive examples and the other two classes for the negative examples. The resulting classifiers are shown.

How do we combine these classifiers to get a prediction on a novel example x? Each of the classifiers outputs some sort of confidence score that it sees a positive example. For instance, with logistic regression, the confidence score is given by $h_\theta(x)$. For decision trees, it is the probability estimate associated with the corresponding leaf node. Our prediction on the new example x will

(a)  (b) 

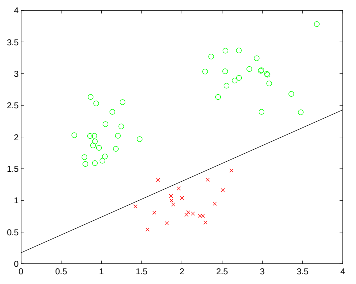
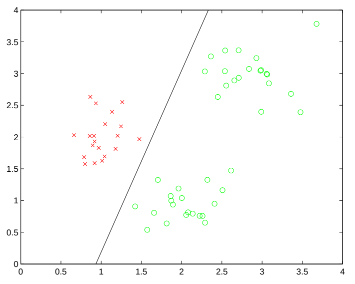
(c)  (d) 

Figure 1: (a) A multiclass classification problem, with three categories. (b-d) Learned classifiers for each of the binary classification subproblems in one-vs.-all.

simply be the class for which the classifier returns the highest confidence score of the example being a member of that class.

2 Bias, variance, and generalization error

In the first machine learning lecture, we introduced the idea of overfitting or underfitting. Recall that we said a model *underfits* the training data if, like the first model in Figure 2 (b), it does not capture all of the structure available from the data. On the other hand, a model *overfits* if it captures too many of the ideosyncrasies of the training data, as in Figure 2 (d). In this section, we define more formally what we mean by overfitting and underfitting.

2.1 Regression

For the moment, let's focus on the regression problem. Suppose we have a training set $S_{\text{train}} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ sampled independently

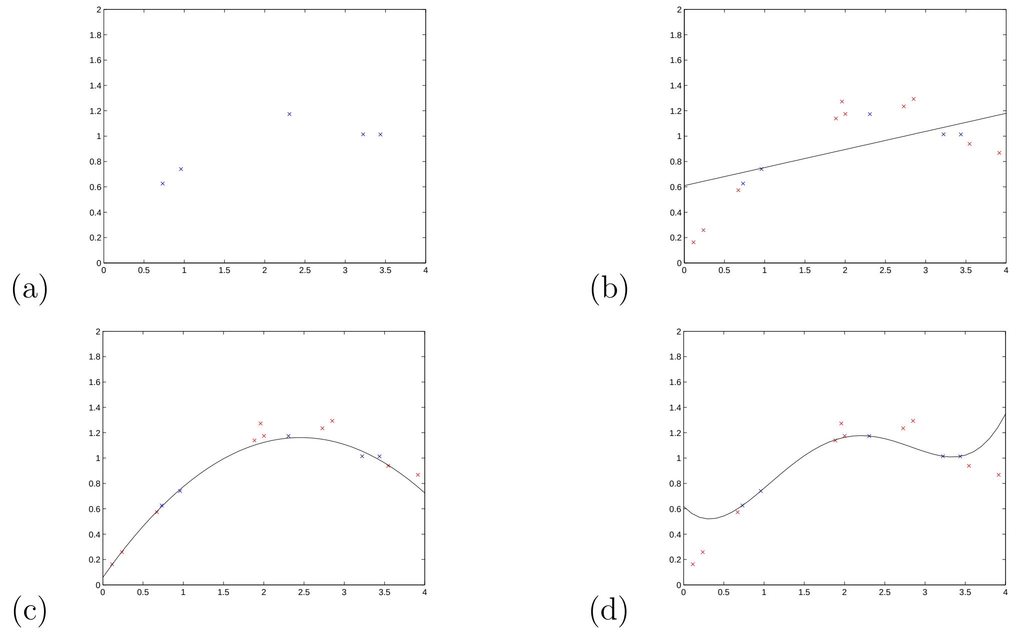


Figure 2: (a) Five data points to which we would like to fit a polynomial model. (b) The linear regression fit of a linear function to the five data points. New test examples are shown in red. (c) The linear regression fit of a quadratic polynomial. (d) The linear regression fit of a fourth-degree polynomial.

and identically distributed (IID) from some distribution $\mathcal{D}$. Define the **average training error** $\varepsilon_{s_{\text{train}}}(h_\theta)$ of a hypothesis h_θ to be:

$$\varepsilon_{S_{\text{train}}}(h_\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

But we're not actually interested in classifying training examples, since we have their labels already. What we care about is the **generalization error**, or the expected error on new examples (drawn from $\mathcal{D}$). This is written as as:

$$\varepsilon(h_\theta) = \mathrm{E}_{(x,y)\,\sim\,\mathcal{D}}[(h_\theta(x) - y)^2].$$

Now we can be more precise about why underfitting and overfitting are bad: they lead to high generalization error. Figure 2 shows what happens when we draw new examples from $\mathcal{D}$. The linear model shown in (b) has high generalization error because it is too simple a model to capture all of the structure in the data. The fourth-order polynomial in (d), however, has high generalization error because it varies too wildly. The best model, the quadratic polynomial in (c), has the lowest generalization error because it captures all the structure in the data, but no more.

We often describe models which underfit as having high **bias**, and models which overfit as having high **variance**. To make sense of these terms, imagine fitting the model parameters to a series of random data sets drawn from $\mathcal{D}$. Such a simulation is shown in Figure 3. The linear model will typically generate roughly the same parameters on each run, and so the "variance" from one trial to the next is small. However, all of these models will be systematically "biased" to underestimate the target values for points in the middle and overestimate the target values of points near the ends. On the other hand, the fourth-degree polynomial estimate varies wildly from trial to trial, and therefore it has high variance. But on average, it tends to guess correctly. This is shown on the right, where the parameters are averaged over many iterations.

We can't directly find out the generalization error, since we only have a finite set of data to work with. Instead, we estimate generalization error using a separate test set $\{(x_{\text{test}}^{(1)}, x_{\text{test}}^{(1)}), \ldots, (x_{\text{test}}^{(k)}, y_{\text{test}}^{(k)})\}$, which we do not include during the training phase. We define the **test error** $\varepsilon_{S_{\text{test}}}$ as follows:

$$\varepsilon_{S_{\text{test}}}(h_\theta) = \sum_{i=1}^{k}(h_\theta(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2.$$

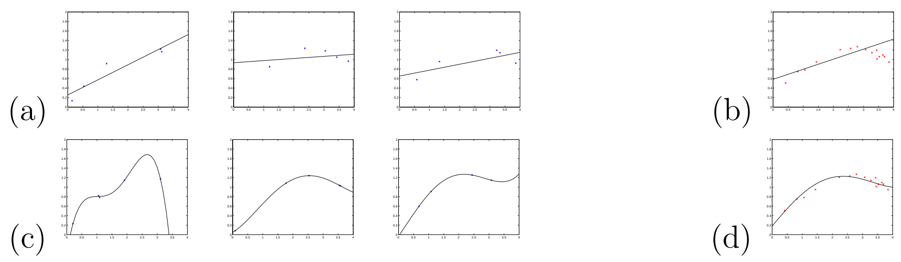


Figure 3: (a) The linear regression fits of linear functions to three different training sets, each uniformly sampled over the interval $[0, 4]$. Because most of these functions look fairly similar, we say the linear model has low variance. (b) The resulting linear model when the parameters are averaged over 50,000 trials. On average, the models systematically underestimate the function in the middle of the range and overestimate it near the ends, so we say the linear model has high bias. (c) The linear regression fits of a fourth-order polynomial to three random training sets. The fourth-order model has high variance. (d) The fourth-order fit averaged over 50,000 trials. The average of all of the parameter values achieves a good fit, so we say the model has low bias.

It can be shown that the test error is a good predictor of the generalization error. Training error is not a good predictor of generalization error; in fact, it will be overly optimistic, especially for more complex models. Recall from the first machine learning lecture our cartoon, shown in Figure 4, which show how training and test error vary as a function of model complexity. (Model complexity might include the degree of the polynomial, the size of the decision tree, or the number of features.)

2.2 Classification

What do overfitting and underfitting mean in the context of classification? Figure 5 shows a cartoon example. Our definitions for classification will be identical to those for regression, except that we use the **0/1 classification error**, the proportion of misclassified examples, rather than mean-squared error, as the penalty. Suppose for a moment that our classifier outputs a binary decision 0 or 1. (This might be achieved, for instance, by choosing a threshold confidence score in logistic regression.) A useful notation is the **indicator function** $1\{\cdot\}$, where $1\{\text{true}\} = 1$ and $1\{\text{false}\} = 0$. For instance, $1\{2 = 3\} = 0$ and $1\{1 + 1 = 2\} = 1$.

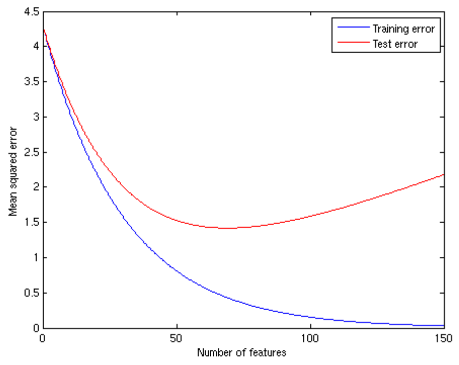


Figure 4: A cartoon picture of training and test error as a function of model complexity. Model complexity increases from left to right.

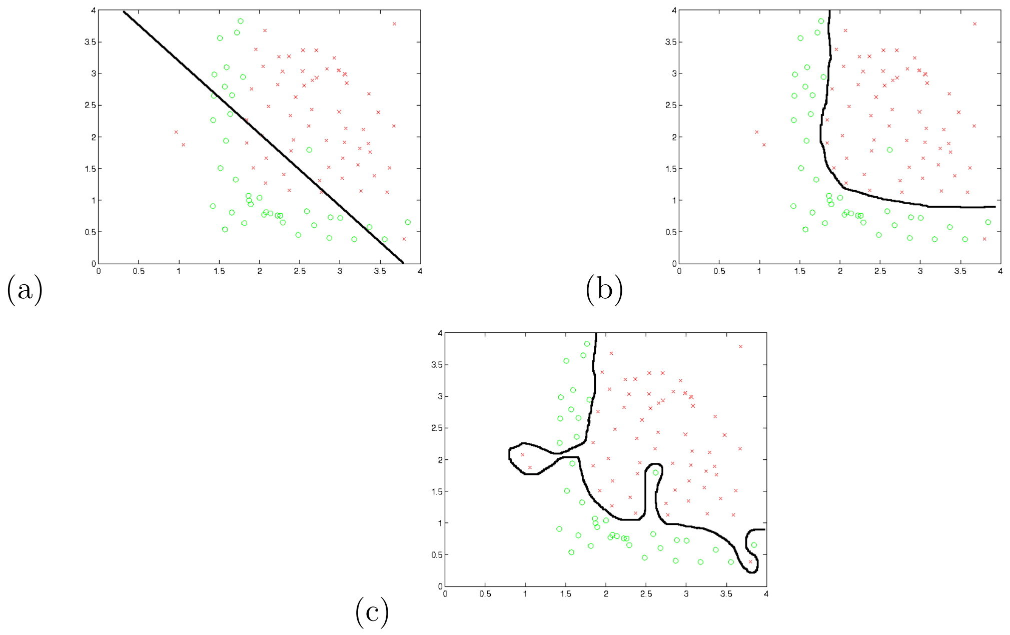


Figure 5: (a) A classifier which underfits the data. (b) A classifier which achieves a good fit. (c) A classifier which overfits.

We define the **training error** as the proportion of training examples which are misclassified:

$$\varepsilon_{S_{\text{train}}}(h_\theta) = \frac{1}{m}\sum_{i=1}^{m} 1\{h_\theta(x^{(i)}_{\text{train}}) \neq y^{(i)}_{\text{train}}\}$$

The **generalization error** is defined as the probability of a new example being misclassified:

$$\varepsilon(h_\theta) = P_{(x,y)\,\sim\,\mathcal{D}}(h_\theta(x) \neq y)$$

All of the properties we outlined above hold true for classification as well as for regression.

3 Bias and variance in practice

Now that we've introduced bias and variance, how do we minimize the problems associated with both of them? In other words, how do we choose a model which achieves a good tradeoff between bias and variance? Often, we might have a fixed set of models to choose from. For instance, we might be choosing between the polynomials of degree i, for $i \leq N$. Or we might be trying to choose the right tree depth for our decision tree classifier. In these cases, it makes sense to use **hold-out cross-validation**. Here, we leave aside part of our training data as a **hold-out cross-validation set** which we use to evaluate the performace of the different models. Specifically, the procedure is as follows:

- Split your data (not including test data) randomly into, say, 70% train and 30% cross-validation.
- For each of your N models, train a hypothesis from the training set. This will result in N different hypotheses.
- Evaluate each of these hypotheses on the cross-validation set. Choose the one with lowest error on the cross-validation set.
- Evaluate this model on the test data. This will give you the final number you report as your estimated generalization error.

Optionally, between the third and fourth step above, you may also retrain the selected model on the combination of the training and cross-validation sets, since training on more data almost always gives better performance.

But, as is the case with most of what we learn, real life is rarely this simple. Often, you won't have a fixed set of models you are trying to choose amongst. Suppose you've trained a learning algorithm and it gives poor generalization error. What do you do next? Use fewer features? More features? Get more data? Implement a different learning algorithm? Some of these options are clearly costly, so it would be nice if we had a rough idea of which ones are likely to work in which situations. Consider the following:

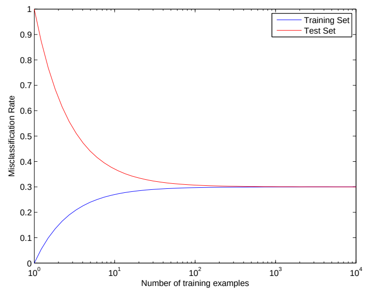
- If your model has high bias, it is too simple. It doesn't fit the structure already there in the data. In this case, you want to make your model more complex, perhaps by adding more features or using deeper decision trees.

- If your model has high variance, it is too complex. It fits the ideosyncratic properties of the data, and doesn't generalize well. In these cases, you want to simplify your hypothesis (e.g. by removing features) or get more data.

Keeping this advice in mind may well save you from wasting six months collecting more data when the problem is high bias, or six months designing new features when the problem is high variance.

How do you actually tell if your problem is bias or variance? You can do so by checking how different the training and test error are. The simplest way to do this is to simply compare the training and test errors given all of your data. If they are very different, your model is likely to be high variance, while if they are almost the same, your model is likely to be high bias. Alternatively, we can also plot the training and test error as a function of the number of training examples. In other words, we train our classifier only on the first 100 examples, then on only the first 200, and so on. Figure 6 shows a cartoon of what such plots might look like if your model is high bias or high variance. If the training and test errors appear to have asymptoted, your model is high bias, while if they are still steadily changing, your model is high variance. (Ask yourself: why does the training error increase as you add more examples?)

More generally, it is impossible to give a general recipe for applying a given machine learning algorithm to any arbitrary problem, since so much depends on particular properties of the problem, such as the amount of data available, the kind of structure in the data, and the noisiness of the data. There is no substitute for a careful analysis of the particular case to determine where the learning algorithm is having trouble.

(a) 

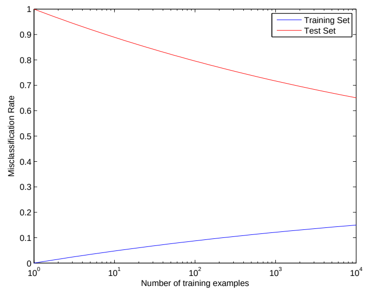
(b) 

Figure 6: Cartoon examples showing how the training and test error might vary with the number of training examples. (a) A model with high bias. (b) A model with high variance.

Actually, this is a special case of a general rule of thumb for building complex AI systems in general. Broadly speaking, there are two ways to approach any problem:

1. Think for a long time about the best way to solve the problem, and then implement it.

2. Implement something quick and dirty to begin with, and then iteratively improve on it as you carefully analyze the deficiencies at each stage.

The second option is often preferable. It is not always a good way to do machine learning research, but it is a good way to get a real-world system working. You almost never know in advance what will go wrong, and unexpected turns can ruin the best-laid plans. You don't know if your model will turn out to be high bias or high variance. Therefore, you don't know in advance whether you should spend time collecting more data or implementing new and better features. Rather, you should build a prototype, check whether the problem is bias or variance, and fix it.

This is analogous to the idea of premature optimization in software engineering. It is hard to predict in advance which part of your code will be the bottleneck. If you try to guess, you will just waste time optimizing sections which make up only a small portion of the running time, and add useless clutter to your code.

Here are some more examples (which you may possibly encounter at some point in your lives) of situations where this rule applies:

- Object recognition. Is the problem that your classifier has a hard time recognizing clocks? Mugs? Or is the problem related to tracking objects, rather than identifying them? Is running time a problem? It is hard to predict any of these factors in advance.

- Robot dog. Does it fall over when it moves its legs? Does its body collide with the terrain? Is the planning algorithm too slow? Is it unable to move its leg from one point to another without hitting a rock?

Build the system first — that will tell you which part of the task is the hard part, and you will know where to focus your attention.

4 Bagged decision trees

We said it's hard to predict in advance whether an algorithm will have a problem with bias or variance, but in fact, single decision tree classifiers famously have a problem with high variance. Why? Recall that models tend to have a problem with variance if their behavior varies wildly depending on the training set. It turns out that our information gain criterion for choosing decision tree split nodes tends to have a lot of "close calls," and because of this, differences in the training data can cause much different trees to be learned.

As we will see shortly, it is possible to reduce the variance of decision trees by training many models on a series of different training sets, and then somehow averaging these models together. In general, algorithms which try to improve performance by averaging different models together are known as **ensemble methods**.

As an intuition for why averaging models should reduce variance, recall the comparisons in Figure 3. We saw that, even though the individual fourth-degree polynomials showed high variance, when we averaged the hypotheses learned from 50,000 different training sets, we got an excellent fit. This is, of course, slightly misleading, because we will never have enough data to generate 50,000 different training sets IID. If we did, we could also improve performace (relative to the individual hypotheses) by combining the 50,000 training sets into one big training set and learning a single hypothesis on the combination. In fact, in the case of linear regression, it turns out we can't do better than simply using all of our training data as one big training set. But surprisingly, in the case of decision trees, we can do better.

A particularly good ensemble method for decision trees is called **bagging**. Suppose we have a training set S_{train} with m examples. We generate a series of B random training sets by sampling m elements from S_{train} *with* *relpacement* (meaning an example may be chosen multiple times). More formally, we use the following procedure:

```
For $b = 1, \ldots, B$,
    For $i = 1, \ldots, m$, choose $(\hat{x}^{(i)}, \hat{y}^{(i)}) \in \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ randomly.
    Learn $h_b$ using $\{(\hat{x}^{(1)}, \hat{y}^{(1)}), \ldots, (\hat{x}^{(m)}, \hat{y}^{(m)})\}$.
```

The final hypothesis $h(x)$ will be the average of the B learned hypotheses, $h(x) = \frac{1}{B}\sum_{b=1}^{B} h_b(x)$.

4.1 Boosting

We just showed how to reduce the variance of decision trees by averaging models learned on random training sets. But we can do even better by systematically choosing the series of training sets to include the "hardest" examples. More specifically, each decision tree is trained on a training set which emphasizes the examples the previous tree got wrong. This general strategy is known as **boosting**, and the particular algorithm we present is **AdaBoost**, or Adaptive Boost. AdaBoost is done as follows:

```
Initialize $D_1(1) = D_1(2) = \cdots = D_1(m) = \frac{1}{m}$.
For $b = 1, \ldots, B$,
    For $i = 1, \ldots, m$,
        Choose $(\hat{x}^{(i)}, \hat{y}^{(i)}) \in \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ randomly, where
        the probability of choosing the $i^{th}$ instance is $D_b(i)$.
    Learn $h_b$ using $\{(\hat{x}^{(1)}, \hat{y}^{(1)}), \ldots, (\hat{x}^{(m)}, \hat{y}^{(m)})\}$.
    Define $\varepsilon_b = \sum_{i=1}^{m} D_b(i) 1\{h_b(x^{(i)}) \neq y^{(i)}\}$.
    Set $\alpha_b = \frac{1}{2}\log\frac{1-\varepsilon_b}{\varepsilon_b}$.
    Set
        $$D_{b+1}(i) = \begin{cases} \frac{D_b(i)e^{-\alpha_b}}{z} & \text{if } h_b(x^{(i)}) = y^{(i)} \\ \frac{D_b(i)e^{\alpha_b}}{z} & \text{if } h_b(x^{(i)}) \neq y^{(i)} \end{cases},$$
    where $z$ is a normalization constant chosen so $\sum_{i=1}^{m} D_{b+1}(i) = 1$.
```

The final hypothesis is given by $h(x) = \sum_{b=1}^{B} h_b(x) \frac{\alpha_b}{\sum_{t=1}^{B} \alpha_t}$.

Boosted decision trees are often regarded as one of the best supervised learning algorithms. They are often used with depth-limited decision trees[1], and the number of decision trees is often very large, on the order of tens of thousands.

There is a remarkable theoretical result about AdaBoost: if each tree is a "weak learner", in that it does slightly better than random chance on a given data set (say it achieves 51% accuracy), the algorithm as a whole will approach zero training error as B approaches ∞.

[1]Sometimes, the depth is limited to 1, so that the decision tree just tests a single feature of the root node, and then makes its prediction. These very small decision trees are also called **decision stumps**.

Generative models and Naive Bayes

9

9.1 CS229 Lecture notes by Andrew Ng, Part IV

CS229 Lecture notes

Andrew Ng

Part IV

Generative Learning algorithms

So far, we've mainly been talking about learning algorithms that model $p(y|x;\theta)$, the conditional distribution of y given x. For instance, logistic regression modeled $p(y|x;\theta)$ as $h_\theta(x) = g(\theta^T x)$ where g is the sigmoid function. In these notes, we'll talk about a different type of learning algorithm.

Consider a classification problem in which we want to learn to distinguish between elephants ($y = 1$) and dogs ($y = 0$), based on some features of an animal. Given a training set, an algorithm like logistic regression or the perceptron algorithm (basically) tries to find a straight line—that is, a decision boundary—that separates the elephants and dogs. Then, to classify a new animal as either an elephant or a dog, it checks on which side of the decision boundary it falls, and makes its prediction accordingly.

Here's a different approach. First, looking at elephants, we can build a model of what elephants look like. Then, looking at dogs, we can build a separate model of what dogs look like. Finally, to classify a new animal, we can match the new animal against the elephant model, and match it against the dog model, to see whether the new animal looks more like the elephants or more like the dogs we had seen in the training set.

Algorithms that try to learn $p(y|x)$ directly (such as logistic regression), or algorithms that try to learn mappings directly from the space of inputs $\mathcal{X}$ to the labels $\{0, 1\}$, (such as the perceptron algorithm) are called **discriminative** learning algorithms. Here, we'll talk about algorithms that instead try to model $p(x|y)$ (and $p(y)$). These algorithms are called **generative** learning algorithms. For instance, if y indicates whether a example is a dog (0) or an elephant (1), then $p(x|y = 0)$ models the distribution of dogs' features, and $p(x|y = 1)$ models the distribution of elephants' features.

After modeling $p(y)$ (called the **class priors**) and $p(x|y)$, our algorithm can then use Bayes rule to derive the posterior distribution on y given x:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}.$$

Here, the denominator is given by $p(x) = p(x|y = 1)p(y = 1) + p(x|y = 0)p(y = 0)$ (you should be able to verify that this is true from the standard properties of probabilities), and thus can also be expressed in terms of the quantities $p(x|y)$ and $p(y)$ that we've learned. Actually, if were calculating $p(y|x)$ in order to make a prediction, then we don't actually need to calculate the denominator, since

$$\begin{aligned} \arg\max_y p(y|x) &= \arg\max_y \frac{p(x|y)p(y)}{p(x)} \\ &= \arg\max_y p(x|y)p(y). \end{aligned}$$

1 Gaussian discriminant analysis

The first generative learning algorithm that we'll look at is Gaussian discriminant analysis (GDA). In this model, we'll assume that $p(x|y)$ is distributed according to a multivariate normal distribution. Lets talk briefly about the properties of multivariate normal distributions before moving on to the GDA model itself.

1.1 The multivariate normal distribution

The multivariate normal distribution in n-dimensions, also called the multivariate Gaussian distribution, is parameterized by a **mean vector** $\mu \in \mathbb{R}^n$ and a **covariance matrix** $\Sigma \in \mathbb{R}^{n\times n}$, where $\Sigma \geq 0$ is symmetric and positive semi-definite. Also written "$\mathcal{N}(\mu, \Sigma)$", its density is given by:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right).$$

In the equation above, "$|\Sigma|$" denotes the determinant of the matrix Σ.

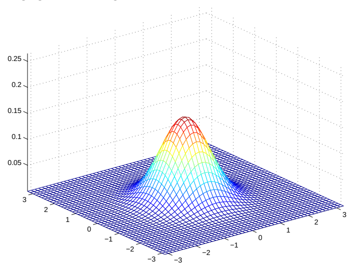
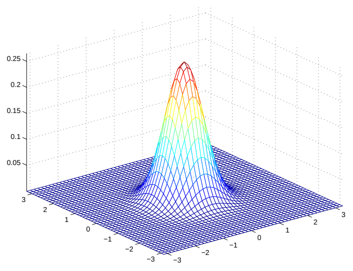
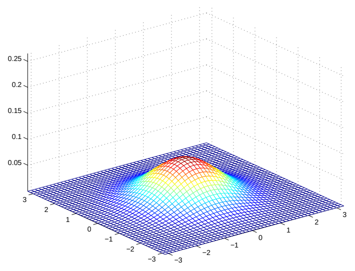
For a random variable X distributed $\mathcal{N}(\mu, \Sigma)$, the mean is (unsurprisingly,) given by μ:

$$\mathrm{E}[X] = \int_x x\, p(x; \mu, \Sigma)dx = \mu$$

The **covariance** of a vector-valued random variable Z is defined as $\mathrm{Cov}(Z) = \mathrm{E}[(Z - \mathrm{E}[Z])(Z - \mathrm{E}[Z])^T]$. This generalizes the notion of the variance of a real-valued random variable. The covariance can also be defined as $\text{Cov}(Z) = \text{E}[ZZ^T] - (\text{E}[Z])(\text{E}[Z])^T$. (You should be able to prove to yourself that these two definitions are equivalent.) If $X \sim \mathcal{N}(\mu, \Sigma)$, then

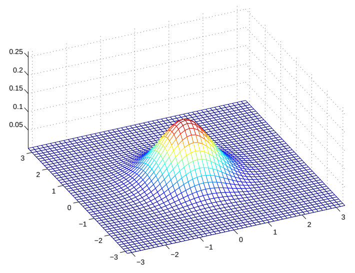
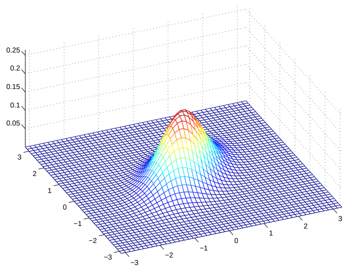
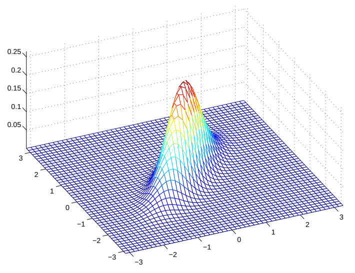
$$\text{Cov}(X) = \Sigma.$$

Here're some examples of what the density of a Gaussian distribution look like:

The left-most figure shows a Gaussian with mean zero (that is, the 2x1 zero-vector) and covariance matrix $\Sigma = I$ (the 2x2 identity matrix). A Gaussian with zero mean and identity covariance is also called the **standard normal distribution**. The middle figure shows the density of a Gaussian with zero mean and $\Sigma = 0.6I$; and in the rightmost figure shows one with , $\Sigma = 2I$. We see that as Σ becomes larger, the Gaussian becomes more "spread-out," and as it becomes smaller, the distribution becomes more "compressed."

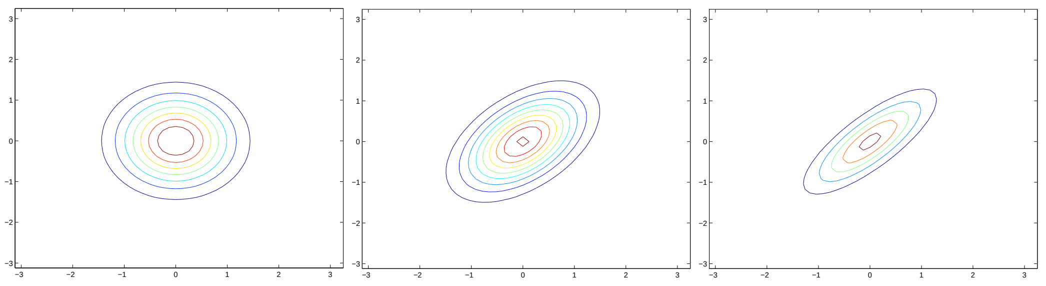
Lets look at some more examples.

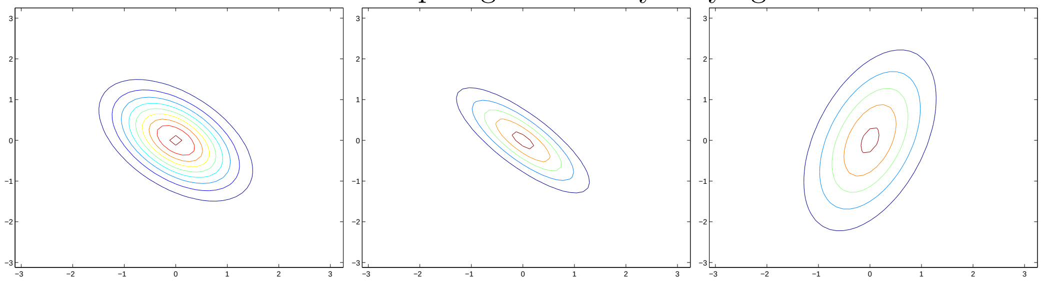
The figures above show Gaussians with mean 0, and with covariance matrices respectively

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}; \quad .\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$

The leftmost figure shows the familiar standard normal distribution, and we see that as we increase the off-diagonal entry in Σ, the density becomes more "compressed" towards the 45° line (given by $x_1 = x_2$). We can see this more clearly when we look at the contours of the same three densities:



Here's one last set of examples generated by varying Σ:

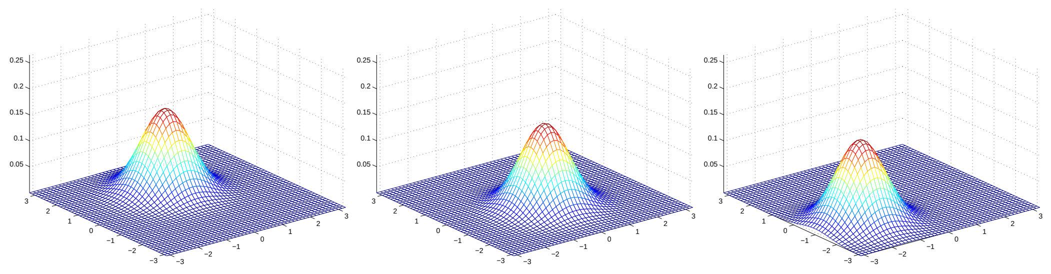


The plots above used, respectively,

$$\Sigma = \begin{bmatrix} 1 & \text{-0.5} \\ \text{-0.5} & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & \text{-0.8} \\ \text{-0.8} & 1 \end{bmatrix}; \quad .\Sigma = \begin{bmatrix} 3 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$

From the leftmost and middle figures, we see that by decreasing the diagonal elements of the covariance matrix, the density now becomes "compressed" again, but in the opposite direction. Lastly, as we vary the parameters, more generally the contours will form ellipses (the rightmost figure showing an example).

As our last set of examples, fixing $\Sigma = I$, by varying μ, we can also move the mean of the density around.



The figures above were generated using $\Sigma = I$, and respectively

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} \text{-0.5} \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} \text{-1} \\ \text{-1.5} \end{bmatrix}.$$

1.2 The Gaussian Discriminant Analysis model

When we have a classification problem in which the input features x are continuous-valued random variables, we can then use the Gaussian Discriminant Analysis (GDA) model, which models $p(x|y)$ using a multivariate normal distribution. The model is:

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi) \\ x|y=0 &\sim \mathcal{N}(\mu_0, \Sigma) \\ x|y=1 &\sim \mathcal{N}(\mu_1, \Sigma) \end{aligned}$$

Writing out the distributions, this is:

$$\begin{aligned} p(y) &= \phi^y(1-\phi)^{1-y} \\ p(x|y=0) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right) \\ p(x|y=1) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right) \end{aligned}$$

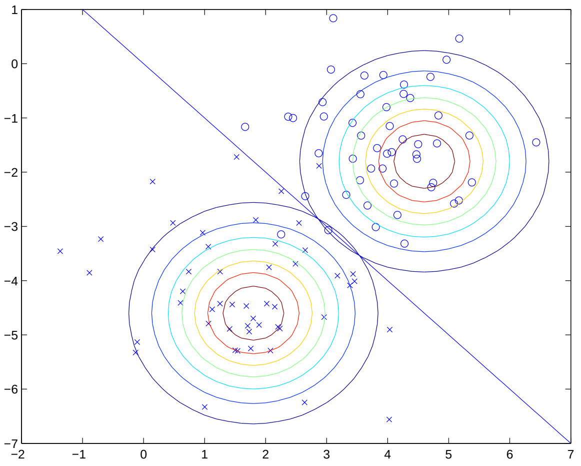
Here, the parameters of our model are ϕ, Σ, μ_0 and μ_1. (Note that while there're two different mean vectors μ_0 and μ_1, this model is usually applied using only one covariance matrix Σ.) The log-likelihood of the data is given by

$$\begin{aligned} \ell(\phi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^{m} p(x^{(i)}|y^{(i)}; \mu_0, \mu_1, \Sigma) p(y^{(i)}; \phi). \end{aligned}$$

By maximizing ℓ with respect to the parameters, we find the maximum likelihood estimate of the parameters (see problem set 1) to be:

$$\begin{aligned}
\phi &= \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\} \\
\mu_0 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \\
\mu_1 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\Sigma &= \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T.
\end{aligned}$$

Pictorially, what the algorithm is doing can be seen in as follows:



Shown in the figure are the training set, as well as the contours of the two Gaussian distributions that have been fit to the data in each of the two classes. Note that the two Gaussians have contours that are the same shape and orientation, since they share a covariance matrix Σ, but they have different means μ_0 and μ_1. Also shown in the figure is the straight line giving the decision boundary at which $p(y = 1|x) = 0.5$. On one side of the boundary, we'll predict $y = 1$ to be the most likely outcome, and on the other side, we'll predict $y = 0$.

1.3 Discussion: GDA and logistic regression

The GDA model has an interesting relationship to logistic regression. If we view the quantity $p(y = 1|x; \phi, \mu_0, \mu_1, \Sigma)$ as a function of x, we'll find that it can be expressed in the form

$$p(y = 1|x; \phi, \Sigma, \mu_0, \mu_1) = \frac{1}{1 + \exp(-\theta^T x)},$$

where θ is some appropriate function of $\phi, \Sigma, \mu_0, \mu_1$.[1] This is exactly the form that logistic regression—a discriminative algorithm—used to model $p(y = 1|x)$.

When would we prefer one model over another? GDA and logistic regression will, in general, give different decision boundaries when trained on the same dataset. Which is better?

We just argued that if $p(x|y)$ is multivariate gaussian (with shared Σ), then $p(y|x)$ necessarily follows a logistic function. The converse, however, is not true; i.e., $p(y|x)$ being a logistic function does not imply $p(x|y)$ is multivariate gaussian. This shows that GDA makes *stronger* modeling assumptions about the data than does logistic regression. It turns out that when these modeling assumptions are correct, then GDA will find better fits to the data, and is a better model. Specifically, when $p(x|y)$ is indeed gaussian (with shared Σ), then GDA is **asymptotically efficient**. Informally, this means that in the limit of very large training sets (large m), there is no algorithm that is strictly better than GDA (in terms of, say, how accurately they estimate $p(y|x)$). In particular, it can be shown that in this setting, GDA will be a better algorithm than logistic regression; and more generally, even for small training set sizes, we would generally expect GDA to better.

In contrast, by making significantly weaker assumptions, logistic regression is also more *robust* and less sensitive to incorrect modeling assumptions. There are many different sets of assumptions that would lead to $p(y|x)$ taking the form of a logistic function. For example, if $x|y = 0 \sim \text{Poisson}(\lambda_0)$, and $x|y = 1 \sim \text{Poisson}(\lambda_1)$, then $p(y|x)$ will be logistic. Logistic regression will also work well on Poisson data like this. But if we were to use GDA on such data—and fit Gaussian distributions to such non-Gaussian data—then the results will be less predictable, and GDA may (or may not) do well.

To summarize: GDA makes stronger modeling assumptions, and is more data efficient (i.e., requires less training data to learn "well") when the modeling assumptions are correct or at least approximately correct. Logistic regression makes weaker assumptions, and is significantly more robust to deviations from modeling assumptions. Specifically, when the data is indeed non-Gaussian, then in the limit of large datasets, logistic regression will

[1]This uses the convention of redefining the $x^{(i)}$'s on the right-hand-side to be $n + 1$-dimensional vectors by adding the extra coordinate $x_0^{(i)} = 1$; see problem set 1.

almost always do better than GDA. For this reason, in practice logistic regression is used more often than GDA. (Some related considerations about discriminative vs. generative models also apply for the Naive Bayes algorithm that we discuss next, but the Naive Bayes algorithm is still considered a very good, and is certainly also a very popular, classification algorithm.)

2 Naive Bayes

In GDA, the feature vectors x were continuous, real-valued vectors. Lets now talk about a different learning algorithm in which the x_i's are discrete-valued.

For our motivating example, consider building an email spam filter using machine learning. Here, we wish to classify messages according to whether they are unsolicited commercial (spam) email, or non-spam email. After learning to do this, we can then have our mail reader automatically filter out the spam messages and perhaps place them in a separate mail folder. Classifying emails is one example of a broader set of problems called **text classification**.

Lets say we have a training set (a set of emails labeled as spam or non-spam). We'll begin our construction of our spam filter by specifying the features x_i used to represent an email.

We will represent an email via a feature vector whose length is equal to the number of words in the dictionary. Specifically, if an email contains the i-th word of the dictionary, then we will set $x_i = 1$; otherwise, we let $x_i = 0$. For instance, the vector

$$x = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \text{a} \\ \text{aardvark} \\ \text{aardwolf} \\ \vdots \\ \text{buy} \\ \vdots \\ \text{zygmurgy} \end{array}$$

is used to represent an email that contains the words "a" and "buy," but not "aardvark," "aardwolf" or "zygmurgy."[2] The set of words encoded into the

[2]Actually, rather than looking through an english dictionary for the list of all english words, in practice it is more common to look through our training set and encode in our feature vector only the words that occur at least once there. Apart from reducing the number of words modeled and hence reducing our computational and space requirements,

feature vector is called the **vocabulary**, so the dimension of x is equal to the size of the vocabulary.

Having chosen our feature vector, we now want to build a discriminative model. So, we have to model $p(x|y)$. But if we have, say, a vocabulary of 50000 words, then $x \in \{0,1\}^{50000}$ (x is a 50000-dimensional vector of 0's and 1's), and if we were to model x explicitly with a multinomial distribution over the 2^{50000} possible outcomes, then we'd end up with a $(2^{50000}-1)$-dimensional parameter vector. This is clearly too many parameters.

To model $p(x|y)$, we will therefore make a very strong assumption. We will assume that the x_i's are conditionally independent given y. This assumption is called the **Naive Bayes (NB) assumption**, and the resulting algorithm is called the **Naive Bayes classifier**. For instance, if $y = 1$ means spam email; "buy" is word 2087 and "price" is word 39831; then we are assuming that if I tell you $y = 1$ (that a particular piece of email is spam), then knowledge of x_{2087} (knowledge of whether "buy" appears in the message) will have no effect on your beliefs about the value of x_{39831} (whether "price" appears). More formally, this can be written $p(x_{2087}|y) = p(x_{2087}|y, x_{39831})$. (Note that this is *not* the same as saying that x_{2087} and x_{39831} are independent, which would have been written "$p(x_{2087}) = p(x_{2087}|x_{39831})$"; rather, we are only assuming that x_{2087} and x_{39831} are conditionally independent *given* y.)

We now have:

$$\begin{aligned}
& p(x_1, \ldots, x_{50000}|y) \\
& \quad = \quad p(x_1|y)p(x_2|y,x_1)p(x_3|y,x_1,x_2)\cdots p(x_{50000}|y,x_1,\ldots,x_{49999}) \\
& \quad = \quad p(x_1|y)p(x_2|y)p(x_3|y)\cdots p(x_{50000}|y) \\
& \quad = \quad \prod_{i=1}^{n} p(x_i|y)
\end{aligned}$$

The first equality simply follows from the usual properties of probabilities, and the second equality used the NB assumption. We note that even though the Naive Bayes assumption is an extremely strong assumptions, the resulting algorithm works well on many problems.

Our model is parameterized by $\phi_{i|y=1} = p(x_i = 1|y = 1)$, $\phi_{i|y=0} = p(x_i = 1|y = 0)$, and $\phi_y = p(y = 1)$. As usual, given a training set $\{(x^{(i)}, y^{(i)}); i =$

this also has the advantage of allowing us to model/include as a feature many words that may appear in your email (such as "cs229") but that you won't find in a dictionary. Sometimes (as in the homework), we also exclude the very high frequency words (which will be words like "the," "of," "and,"; these high frequency, "content free" words are called **stop words**) since they occur in so many documents and do little to indicate whether an email is spam or non-spam.

$1, \dots, m\}$, we can write down the joint likelihood of the data:

$$\mathcal{L}(\phi_y, \phi_{i|y=0}, \phi_{i|y=1}) = \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}).$$

Maximizing this with respect to $\phi_y, \phi_{i|y=0}$ and $\phi_{i|y=1}$ gives the maximum likelihood estimates:

$$\begin{aligned}
\phi_{j|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\phi_{j|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \\
\phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}
\end{aligned}$$

In the equations above, the "$\wedge$" symbol means "and." The parameters have a very natural interpretation. For instance, $\phi_{j|y=1}$ is just the fraction of the spam ($y = 1$) emails in which word j does appear.

Having fit all these parameters, to make a prediction on a new example with features x, we then simply calculate

$$\begin{aligned}
p(y = 1|x) &= \frac{p(x|y = 1)p(y = 1)}{p(x)} \\
&= \frac{\left(\prod_{i=1}^{n} p(x_i|y = 1)\right) p(y = 1)}{\left(\prod_{i=1}^{n} p(x_i|y = 1)\right) p(y = 1) + \left(\prod_{i=1}^{n} p(x_i|y = 0)\right) p(y = 0)},
\end{aligned}$$

and pick whichever class has the higher posterior probability.

Lastly, we note that while we have developed the Naive Bayes algorithm mainly for the case of problems where the features x_i are binary-valued, the generalization to where x_i can take values in $\{1, 2, \dots, k_i\}$ is straightforward. Here, we would simply model $p(x_i|y)$ as multinomial rather than as Bernoulli. Indeed, even if some original input attribute (say, the living area of a house, as in our earlier example) were continuous valued, it is quite common to **discretize** it—that is, turn it into a small set of discrete values—and apply Naive Bayes. For instance, if we use some feature x_i to represent living area, we might discretize the continuous values as follows:

Living area (sq. feet)	< 400	400-800	800-1200	1200-1600	>1600
x_i	1	2	3	4	5

Thus, for a house with living area 890 square feet, we would set the value of the corresponding feature x_i to 3. We can then apply the Naive Bayes algorithm, and model $p(x_i|y)$ with a multinomial distribution, as described previously. When the original, continuous-valued attributes are not well-modeled by a multivariate normal distribution, discretizing the features and using Naive Bayes (instead of GDA) will often result in a better classifier.

2.1 Laplace smoothing

The Naive Bayes algorithm as we have described it will work fairly well for many problems, but there is a simple change that makes it work much better, especially for text classification. Lets briefly discuss a problem with the algorithm in its current form, and then talk about how we can fix it.

Consider spam/email classification, and lets suppose that, after completing CS229 and having done excellent work on the project, you decide around June 2003 to submit the work you did to the NIPS conference for publication. (NIPS is one of the top machine learning conferences, and the deadline for submitting a paper is typically in late June or early July.) Because you end up discussing the conference in your emails, you also start getting messages with the word "nips" in it. But this is your first NIPS paper, and until this time, you had not previously seen any emails containing the word "nips"; in particular "nips" did not ever appear in your training set of spam/non-spam emails. Assuming that "nips" was the 35000th word in the dictionary, your Naive Bayes spam filter therefore had picked its maximum likelihood estimates of the parameters $\phi_{35000|y}$ to be

$$\begin{aligned}\phi_{35000|y=1} &= \frac{\sum_{i=1}^m 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}} = 0 \\ \phi_{35000|y=0} &= \frac{\sum_{i=1}^m 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}} = 0\end{aligned}$$

I.e., because it has never seen "nips" before in either spam or non-spam training examples, it thinks the probability of seeing it in either type of email is zero. Hence, when trying to decide if one of these messages containing "nips" is spam, it calculates the class posterior probabilities, and obtains

$$\begin{aligned}p(y=1|x) &= \frac{\prod_{i=1}^n p(x_i|y=1)p(y=1)}{\prod_{i=1}^n p(x_i|y=1)p(y=1) + \prod_{i=1}^n p(x_i|y=0)p(y=0)} \\ &= \frac{0}{0}.\end{aligned}$$

This is because each of the terms "$\prod_{i=1}^n p(x_i|y)$" includes a term $p(x_{35000}|y) = 0$ that is multiplied into it. Hence, our algorithm obtains $0/0$, and doesn't know how to make a prediction.

Stating the problem more broadly, it is statistically a bad idea to estimate the probability of some event to be zero just because you haven't seen it before in your finite training set. Take the problem of estimating the mean of a multinomial random variable z taking values in $\{1, \ldots, k\}$. We can parameterize our multinomial with $\phi_i = p(z = i)$. Given a set of m independent observations $\{z^{(1)}, \ldots, z^{(m)}\}$, the maximum likelihood estimates are given by

$$\phi_j = \frac{\sum_{i=1}^m 1\{z^{(i)} = j\}}{m}.$$

As we saw previously, if we were to use these maximum likelihood estimates, then some of the ϕ_j's might end up as zero, which was a problem. To avoid this, we can use **Laplace smoothing**, which replaces the above estimate with

$$\phi_j = \frac{\sum_{i=1}^m 1\{z^{(i)} = j\} + 1}{m + k}.$$

Here, we've added 1 to the numerator, and k to the denominator. Note that $\sum_{j=1}^k \phi_j = 1$ still holds (check this yourself!), which is a desirable property since the ϕ_j's are estimates for probabilities that we know must sum to 1. Also, $\phi_j \neq 0$ for all values of j, solving our problem of probabilities being estimated as zero. Under certain (arguably quite strong) conditions, it can be shown that the Laplace smoothing actually gives the optimal estimator of the ϕ_j's.

Returning to our Naive Bayes classifier, with Laplace smoothing, we therefore obtain the following estimates of the parameters:

$$\begin{aligned}
\phi_{j|y=1} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 1\} + 2} \\
\phi_{j|y=0} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 0\} + 2}
\end{aligned}$$

(In practice, it usually doesn't matter much whether we apply Laplace smoothing to ϕ_y or not, since we will typically have a fair fraction each of spam and non-spam messages, so ϕ_y will be a reasonable estimate of $p(y = 1)$ and will be quite far from 0 anyway.)

2.2 Event models for text classification

To close off our discussion of generative learning algorithms, lets talk about one more model that is specifically for text classification. While Naive Bayes as we've presented it will work well for many classification problems, for text classification, there is a related model that does even better.

In the specific context of text classification, Naive Bayes as presented uses the what's called the **multi-variate Bernoulli event model**. In this model, we assumed that the way an email is generated is that first it is randomly determined (according to the class priors $p(y)$) whether a spammer or non-spammer will send you your next message. Then, the person sending the email runs through the dictionary, deciding whether to include each word i in that email independently and according to the probabilities $p(x_i = 1|y) = \phi_{i|y}$. Thus, the probability of a message was given by $p(y)\prod_{i=1}^{n} p(x_i|y)$.

Here's a different model, called the **multinomial event model**. To describe this model, we will use a different notation and set of features for representing emails. We let x_i denote the identity of the i-th word in the email. Thus, x_i is now an integer taking values in $\{1, \ldots, |V|\}$, where $|V|$ is the size of our vocabulary (dictionary). An email of n words is now represented by a vector $(x_1, x_2, \ldots, x_n)$ of length n; note that n can vary for different documents. For instance, if an email starts with "A NIPS ...," then $x_1 = 1$ ("a" is the first word in the dictionary), and $x_2 = 35000$ (if "nips" is the 35000th word in the dictionary).

In the multinomial event model, we assume that the way an email is generated is via a random process in which spam/non-spam is first determined (according to $p(y)$) as before. Then, the sender of the email writes the email by first generating x_1 from some multinomial distribution over words ($p(x_1|y)$). Next, the second word x_2 is chosen independently of x_1 but from the same multinomial distribution, and similarly for x_3, x_4, and so on, until all n words of the email have been generated. Thus, the overall probability of a message is given by $p(y)\prod_{i=1}^{n} p(x_i|y)$. Note that this formula looks like the one we had earlier for the probability of a message under the multi-variate Bernoulli event model, but that the terms in the formula now mean very different things. In particular $x_i|y$ is now a multinomial, rather than a Bernoulli distribution.

The parameters for our new model are $\phi_y = p(y)$ as before, $\phi_{i|y=1} = p(x_j = i|y = 1)$ (for any j) and $\phi_{i|y=0} = p(x_j = i|y = 0)$. Note that we have assumed that $p(x_j|y)$ is the same for all values of j (i.e., that the distribution according to which a word is generated does not depend on its position j within the email).

If we are given a training set $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ where $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_{n_i}^{(i)})$ (here, n_i is the number of words in the i-training example), the likelihood of the data is given by

$$\begin{aligned}\mathcal{L}(\phi, \phi_{i|y=0}, \phi_{i|y=1}) &= \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}) \\ &= \prod_{i=1}^{m} \left(\prod_{j=1}^{n_i} p(x_j^{(i)}|y; \phi_{i|y=0}, \phi_{i|y=1}) \right) p(y^{(i)}; \phi_y).\end{aligned}$$

Maximizing this yields the maximum likelihood estimates of the parameters:

$$\begin{aligned}\phi_{k|y=1} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i} \\ \phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}.\end{aligned}$$

If we were to apply Laplace smoothing (which needed in practice for good performance) when estimating $\phi_{k|y=0}$ and $\phi_{k|y=1}$, we add 1 to the numerators and $|V|$ to the denominators, and obtain:

$$\begin{aligned}\phi_{k|y=1} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i + |V|} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i + |V|}.\end{aligned}$$

While not necessarily the very best classification algorithm, the Naive Bayes classifier often works surprisingly well. It is often also a very good "first thing to try," given its simplicity and ease of implementation.

Unsupervised learning

10

10.1 K-maens clustering

CS229 Lecture notes by Andrew Ng

10.2 Mixture of Gaussians and EM algorithm

CS229 Lecture notes by Andrew Ng

10.3 The Boltzmann and Helmholts machines

CS229 Lecture notes

Andrew Ng

The k-means clustering algorithm

In the clustering problem, we are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$, and want to group the data into a few cohesive "clusters." Here, $x^{(i)} \in \mathbb{R}^n$ as usual; but no labels $y^{(i)}$ are given. So, this is an unsupervised learning problem.

The k-means clustering algorithm is as follows:

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {

```
    For every $i$, set
        $c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$
    For each $j$, set
        $\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$
}
```

In the algorithm above, k (a parameter of the algorithm) is the number of clusters we want to find; and the cluster centroids μ_j represent our current guesses for the positions of the centers of the clusters. To initialize the cluster centroids (in step 1 of the algorithm above), we could choose k training examples randomly, and set the cluster centroids to be equal to the values of these k examples. (Other initialization methods are also possible.)

The inner-loop of the algorithm repeatedly carries out two steps: (i) "Assigning" each training example $x^{(i)}$ to the closest cluster centroid μ_j, and (ii) Moving each cluster centroid μ_j to the mean of the points assigned to it. Figure 1 shows an illustration of running k-means.

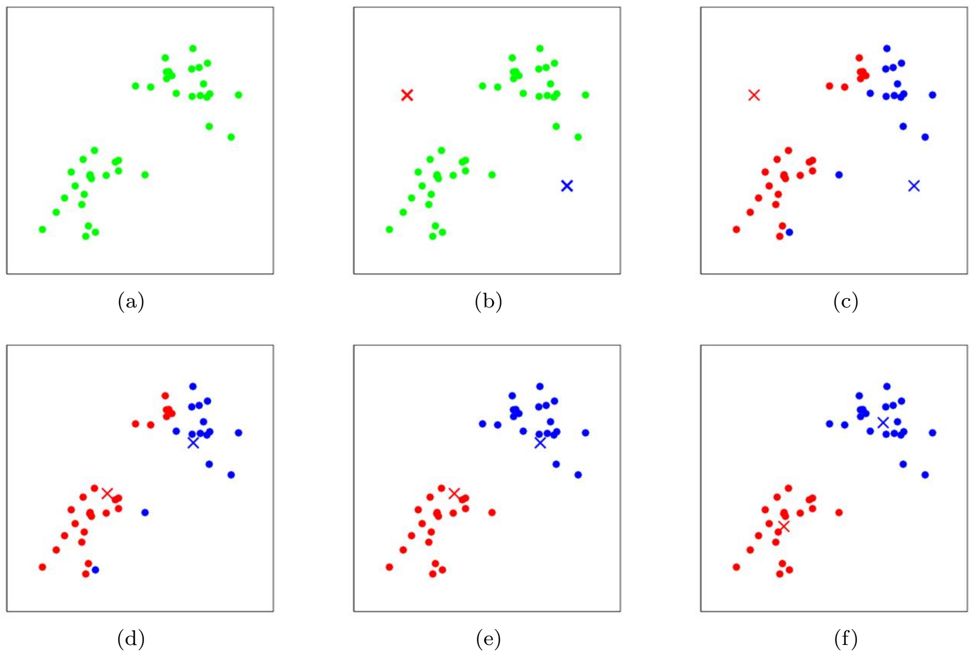


Figure 1: K-means algorithm. Training examples are shown as dots, and cluster centroids are shown as crosses. (a) Original dataset. (b) Random initial cluster centroids (in this instance, not chosen to be equal to two training examples). (c-f) Illustration of running two iterations of k-means. In each iteration, we assign each training example to the closest cluster centroid (shown by "painting" the training examples the same color as the cluster centroid to which is assigned); then we move each cluster centroid to the mean of the points assigned to it. (Best viewed in color.) Images courtesy Michael Jordan.

Is the k-means algorithm guaranteed to converge? Yes it is, in a certain sense. In particular, let us define the **distortion function** to be:

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Thus, J measures the sum of squared distances between each training example $x^{(i)}$ and the cluster centroid $\mu_{c^{(i)}}$ to which it has been assigned. It can be shown that k-means is exactly coordinate descent on J. Specifically, the inner-loop of k-means repeatedly minimizes J with respect to c while holding μ fixed, and then minimizes J with respect to μ while holding c fixed. Thus, J must monotonically decrease, and the value of J must converge. (Usually, this implies that c and μ will converge too. In theory, it is possible for

k-means to oscillate between a few different clusterings—i.e., a few different values for c and/or μ—that have exactly the same value of J, but this almost never happens in practice.)

The distortion function J is a non-convex function, and so coordinate descent on J is not guaranteed to converge to the global minimum. In other words, k-means can be susceptible to local optima. Very often k-means will work fine and come up with very good clusterings despite this. But if you are worried about getting stuck in bad local minima, one common thing to do is run k-means many times (using different random initial values for the cluster centroids μ_j). Then, out of all the different clusterings found, pick the one that gives the lowest distortion $J(c, \mu)$.

CS229 Lecture notes

Andrew Ng

Mixtures of Gaussians and the EM algorithm

In this set of notes, we discuss the EM (Expectation-Maximization) for density estimation.

Suppose that we are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ as usual. Since we are in the unsupervised learning setting, these points do not come with any labels.

We wish to model the data by specifying a joint distribution $p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)})p(z^{(i)})$. Here, $z^{(i)} \sim \text{Multinomial}(\phi)$ (where $\phi_j \geq 0$, $\sum_{j=1}^k \phi_j = 1$, and the parameter ϕ_j gives $p(z^{(i)} = j)$,), and $x^{(i)}|z^{(i)} = j \sim \mathcal{N}(\mu_j, \Sigma_j)$. We let k denote the number of values that the $z^{(i)}$'s can take on. Thus, our model posits that each $x^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $\{1, \ldots, k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depeneding on $z^{(i)}$. This is called the **mixture of Gaussians** model. Also, note that the $z^{(i)}$'s are **latent** random variables, meaning that they're hidden/unobserved. This is what will make our estimation problem difficult.

The parameters of our model are thus ϕ, ϕ and Σ. To estimate them, we can write down the likelihood of our data:

$$\begin{aligned}
\ell(\phi, \mu, \Sigma) &= \sum_{i=1}^m \log p(x^{(i)}; \phi, \mu, \Sigma) \\
&= \sum_{i=1}^m \log \sum_{z^{(i)}=1}^k p(x^{(i)}|z^{(i)}; \mu, \Sigma) p(z^{(i)}; \phi).
\end{aligned}$$

However, if we set to zero the derivatives of this formula with respect to the parameters and try to solve, we'll find that it is not possible to find the maximum likelihood estimates of the parameters in closed form. (Try this yourself at home.)

The random variables $z^{(i)}$ indicate which of the k Gaussians each $x^{(i)}$ had come from. Note that if we knew what the $z^{(i)}$'s were, the maximum likelihood problem would have been easy. Specifically, we could then write down the likelihood as

$$\ell(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \log p(x^{(i)}|z^{(i)}; \mu, \Sigma) + \log p(z^{(i)}; \phi).$$

Maximizing this with respect to ϕ, μ and Σ gives the parameters:

$$\begin{aligned}
\phi_j &= \frac{1}{m}\sum_{i=1}^{m} 1\{z^{(i)} = j\}, \\
\mu_j &= \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\}x^{(i)}}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}, \\
\Sigma_j &= \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\}(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}.
\end{aligned}$$

Indeed, we see that if the $z^{(i)}$'s were known, then maximum likelihood estimation becomes nearly identical to what we had when estimating the parameters of the Gaussian discriminant analysis model, except that here the $z^{(i)}$'s playing the role of the class labels.[1]

However, in our density estimation problem, the $z^{(i)}$'s are *not* known. What can we do?

The EM algorithm is an iterative algorithm that has two main steps. Applied to our problem, in the E-step, it tries to "guess" the values of the $z^{(i)}$'s. In the M-step, it updates the parameters of our model based on our guesses. Since in the M-step we are pretending that the guesses in the first part were correct, the maximization becomes easy. Here's the algorithm:

Repeat until convergence: {

(E-step) For each i, j, set

$$w_j^{(i)} := p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$$

[1]There are other minor differences in the formulas here from what we'd obtained in PS1 with Gaussian discriminant analysis, first because we've generalized the $z^{(i)}$'s to be multinomial rather than Bernoulli, and second because here we are using a different Σ_j for each Gaussian.

(M-step) Update the parameters:

$$\begin{aligned}\phi_j &:= \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)},\\ \mu_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}},\\ \Sigma_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)}-\mu_j)(x^{(i)}-\mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}}\end{aligned}$$

}

In the E-step, we calculate the posterior probability of our parameters the $z^{(i)}$'s, given the $x^{(i)}$ and using the current setting of our parameters. I.e., using Bayes rule, we obtain:

$$p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma) = \frac{p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)p(z^{(i)} = j; \phi)}{\sum_{l=1}^{k} p(x^{(i)}|z^{(i)} = l; \mu, \Sigma)p(z^{(i)} = l; \phi)}$$

Here, $p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)$ is given by evaluating the density of a Gaussian with mean μ_j and covariance Σ_j at $x^{(i)}$; $p(z^{(i)} = j; \phi)$ is given by ϕ_j, and so on. The values $w_j^{(i)}$ calculated in the E-step represent our "soft" guesses[2] for the values of $z^{(i)}$.

Also, you should contrast the updates in the M-step with the formulas we had when the $z^{(i)}$'s were known exactly. They are identical, except that instead of the indicator functions "$1\{z^{(i)} = j\}$" indicating from which Gaussian each datapoint had come, we now instead have the $w_j^{(i)}$'s.

The EM-algorithm is also reminiscent of the K-means clustering algorithm, except that instead of the "hard" cluster assignments $c(i)$, we instead have the "soft" assignments $w_j^{(i)}$. Similar to K-means, it is also susceptible to local optima, so reinitializing at several different initial parameters may be a good idea.

It's clear that the EM algorithm has a very natural interpretation of repeatedly trying to guess the unknown $z^{(i)}$'s; but how did it come about, and can we make any guarantees about it, such as regarding its convergence? In the next set of notes, we will describe a more general view of EM, one

[2]The term "soft" refers to our guesses being probabilities and taking values in $[0, 1]$; in contrast, a "hard" guess is one that represents a single best guess (such as taking values in $\{0, 1\}$ or $\{1, \ldots, k\}$).

that will allow us to easily apply it to other estimation problems in which there are also latent variables, and which will allow us to give a convergence guarantee.

CS229 Lecture notes

Andrew Ng

Part IX
The EM algorithm

In the previous set of notes, we talked about the EM algorithm as applied to fitting a mixture of Gaussians. In this set of notes, we give a broader view of the EM algorithm, and show how it can be applied to a large family of estimation problems with latent variables. We begin our discussion with a very useful result called **Jensen's inequality**

1 Jensen's inequality

Let f be a function whose domain is the set of real numbers. Recall that f is a convex function if $f''(x) \geq 0$ (for all $x \in \mathbb{R}$). In the case of f taking vector-valued inputs, this is generalized to the condition that its hessian H is positive semi-definite ($H \geq 0$). If $f''(x) > 0$ for all x, then we say f is **strictly** convex (in the vector-valued case, the corresponding statement is that H must be strictly positive semi-definite, written $H > 0$). Jensen's inequality can then be stated as follows:

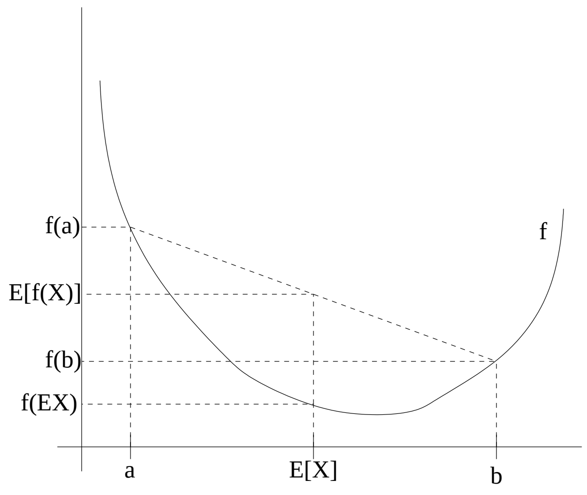
Theorem. Let f be a convex function, and let X be a random variable. Then:

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}X).$$

Moreover, if f is strictly convex, then $\mathrm{E}[f(X)] = f(\mathrm{E}X)$ holds true if and only if $X = \mathrm{E}[X]$ with probability 1 (i.e., if X is a constant).

Recall our convention of occasionally dropping the parentheses when writing expectations, so in the theorem above, $f(\mathrm{E}X) = f(\mathrm{E}[X])$.

For an interpretation of the theorem, consider the figure below.



Here, f is a convex function shown by the solid line. Also, X is a random variable that has a 0.5 chance of taking the value a, and a 0.5 chance of taking the value b (indicated on the x-axis). Thus, the expected value of X is given by the midpoint between a and b.

We also see the values $f(a)$, $f(b)$ and $f(\mathrm{E}[X])$ indicated on the y-axis. Moreover, the value $\mathrm{E}[f(X)]$ is now the midpoint on the y-axis between $f(a)$ and $f(b)$. From our example, we see that because f is convex, it must be the case that $\mathrm{E}[f(X)] \geq f(\mathrm{E}X)$.

Incidentally, quite a lot of people have trouble remembering which way the inequality goes, and remembering a picture like this is a good way to quickly figure out the answer.

Remark. Recall that f is [strictly] concave if and only if $-f$ is [strictly] convex (i.e., $f''(x) \leq 0$ or $H \leq 0$). Jensen's inequality also holds for concave functions f, but with the direction of all the inequalities reversed ($\mathrm{E}[f(X)] \leq f(\mathrm{E}X)$, etc.).

2 The EM algorithm

Suppose we have an estimation problem in which we have a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ consisting of m independent examples. We wish to fit the parameters of a model $p(x, z)$ to the data, where the likelihood is given by

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^{m} \log p(x;\theta) \\ &= \sum_{i=1}^{m} \log \sum_{z} p(x, z; \theta). \end{aligned}$$

But, explicitly finding the maximum likelihood estimates of the parameters θ may be hard. Here, the $z^{(i)}$'s are the latent random variables; and it is often the case that if the $z^{(i)}$'s were observed, then maximum likelihood estimation would be easy.

In such a setting, the EM algorithm gives an efficient method for maximum likelihood estimation. Maximizing $\ell(\theta)$ explicitly might be difficult, and our strategy will be to instead repeatedly construct a lower-bound on ℓ (E-step), and then optimize that lower-bound (M-step).

For each i, let Q_i be some distribution over the z's ($\sum_z Q_i(z) = 1$, $Q_i(z) \geq 0$). Consider the following:[1]

$$\begin{aligned}
\sum_i \log p(x^{(i)};\theta) &= \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)};\theta) && (1)\\
&= \sum_i \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})} && (2)\\
&\geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})} && (3)
\end{aligned}$$

The last step of this derivation used Jensen's inequality. Specifically, $f(x) = \log x$ is a concave function, since $f''(x) = -1/x^2 < 0$ over its domain $x \in \mathbb{R}^+$. Also, the term

$$\sum_{z^{(i)}} Q_i(z^{(i)}) \left[\frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right]$$

in the summation is just an expectation of the quantity $\left[p(x^{(i)}, z^{(i)};\theta)/Q_i(z^{(i)})\right]$ with respect to $z^{(i)}$ drawn according to the distribution given by Q_i. By Jensen's inequality, we have

$$f\left(\mathrm{E}_{z^{(i)}\sim Q_i}\left[\frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right]\right) \geq \mathrm{E}_{z^{(i)}\sim Q_i}\left[f\left(\frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right)\right],$$

where the "$z^{(i)} \sim Q_i$" subscripts above indicate that the expectations are with respect to $z^{(i)}$ drawn from Q_i. This allowed us to go from Equation (2) to Equation (3).

Now, for *any* set of distributions Q_i, the formula (3) gives a lower-bound on $\ell(\theta)$. There're many possible choices for the Q_i's. Which should we choose? Well, if we have some current guess θ of the parameters, it seems

[1]If z were continuous, then Q_i would be a density, and the summations over z in our discussion are replaced with integrals over z.

natural to try to make the lower-bound tight at that value of θ. I.e., we'll make the inequality above hold with equality at our particular value of θ. (We'll see later how this enables us to prove that $\ell(\theta)$ increases monotonically with successsive iterations of EM.)

To make the bound tight for a particular value of θ, we need for the step involving Jensen's inequality in our derivation above to hold with equality. For this to be true, we know it is sufficient that that the expectation be taken over a "constant"-valued random variable. I.e., we require that

$$\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} = c$$

for some constant c that does not depend on $z^{(i)}$. This is easily accomplished by choosing

$$Q_i(z^{(i)}) \propto p(x^{(i)}, z^{(i)}; \theta).$$

Actually, since we know $\sum_z Q_i(z^{(i)}) = 1$ (because it is a distribution), this further tells us that

$$\begin{aligned}
Q_i(z^{(i)}) &= \frac{p(x^{(i)}, z^{(i)}; \theta)}{\sum_z p(x^{(i)}, z; \theta)} \\
&= \frac{p(x^{(i)}, z^{(i)}; \theta)}{p(x^{(i)}; \theta)} \\
&= p(z^{(i)}|x^{(i)}; \theta)
\end{aligned}$$

Thus, we simply set the Q_i's to be the posterior distribution of the $z^{(i)}$'s given $x^{(i)}$ and the setting of the parameters θ.

Now, for this choice of the Q_i's, Equation (3) gives a lower-bound on the loglikelihood ℓ that we're trying to maximize. This is the E-step. In the M-step of the algorithm, we then maximize our formula in Equation (3) with respect to the parameters to obtain a new setting of the θ's. Repeatedly carrying out these two steps gives us the EM algorithm, which is as follows:

```
Repeat until convergence {
    (E-step) For each $i$, set
        $Q_i(z^{(i)}) := p(z^{(i)}|x^{(i)}; \theta).$
    (M-step) Set
        $\theta := \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.$
```

}

How we we know if this algorithm will converge? Well, suppose $\theta^{(t)}$ and $\theta^{(t+1)}$ are the parameters from two successive iterations of EM. We will now prove that $\ell(\theta^{(t)}) \leq \ell(\theta^{(t+1)})$, which shows EM always monotonically improves the log-likelihood. The key to showing this result lies in our choice of the Q_i's. Specifically, on the iteration of EM in which the parameters had started out as $\theta^{(t)}$, we would have chosen $Q_i^{(t)}(z^{(i)}) := p(z^{(i)}|x^{(i)};\theta^{(t)})$. We saw earlier that this choice ensures that Jensen's inequality, as applied to get Equation (3), holds with equality, and hence

$$\ell(\theta^{(t)}) = \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t)})}{Q_i^{(t)}(z^{(i)})}.$$

The parameters $\theta^{(t+1)}$ are then obtained by maximizing the right hand side of the equation above. Thus,

$$\begin{aligned}
\ell(\theta^{(t+1)}) &\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t+1)})}{Q_i^{(t)}(z^{(i)})} && (4) \\
&\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t)})}{Q_i^{(t)}(z^{(i)})} && (5) \\
&= \ell(\theta^{(t)}) && (6)
\end{aligned}$$

This first inequality comes from the fact that

$$\ell(\theta) \geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

holds for any values of Q_i and θ, and in particular holds for $Q_i = Q_i^{(t)}$, $\theta = \theta^{(t+1)}$. To get Equation (5), we used the fact that $\theta^{(t+1)}$ is chosen explicitly to be

$$\arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})},$$

and thus this formula evaluated at $\theta^{(t+1)}$ must be equal to or larger than the same formula evaluated at $\theta^{(t)}$. Finally, the step used to get (6) was shown earlier, and follows from $Q_i^{(t)}$ having been chosen to make Jensen's inequality hold with equality at $\theta^{(t)}$.

Hence, EM causes the likelihood to converge monotonically. In our description of the EM algorithm, we said we'd run it until convergence. Given the result that we just showed, one reasonable convergence test would be to check if the increase in $\ell(\theta)$ between successive iterations is smaller than some tolerance parameter, and to declare convergence if EM is improving $\ell(\theta)$ too slowly.

Remark. If we define

$$J(Q,\theta) = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})},$$

the we know $\ell(\theta) \geq J(Q,\theta)$ from our previous derivation. The EM can also be viewed a coordinate ascent on J, in which the E-step maximizes it with respect to Q (check this yourself), and the M-step maximizes it with respect to θ.

3 Mixture of Gaussians revisited

Armed with our general definition of the EM algorithm, lets go back to our old example of fitting the parameters ϕ, μ and Σ in a mixture of Gaussians. For the sake of brevity, we carry out the derivations for the M-step updates only for ϕ and μ_j, and leave the updates for Σ_j as an exercise for the reader.

The E-step is easy. Following our algorithm derivation above, we simply calculate

$$w_j^{(i)} = Q_i(z^{(i)} = j) = P(z^{(i)} = j | x^{(i)}; \phi, \mu, \Sigma).$$

Here, "$Q_i(z^{(i)} = j)$" denotes the probability of $z^{(i)}$ taking the value j under the distribution Q_i.

Next, in the M-step, we need to maximize, with respect to our parameters ϕ, μ, Σ, the quantity

$$\begin{aligned}
&\sum_{i=1}^m \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, \Sigma)}{Q_i(z^{(i)})} \\
&\quad = \sum_{i=1}^m \sum_{j=1}^k Q_i(z^{(i)} = j) \log \frac{p(x^{(i)} | z^{(i)} = j; \mu, \Sigma) p(z^{(i)} = j; \phi)}{Q_i(z^{(i)} = j)} \\
&\quad = \sum_{i=1}^m \sum_{j=1}^k w_j^{(i)} \log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)\right) \cdot \phi_j}{w_j^{(i)}}
\end{aligned}$$

Lets maximize this with respect to μ_l. If we take the derivative with respect to μ_l, we find

$$\begin{aligned}
&\nabla_{\mu_l} \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)}-\mu_j)^T \Sigma_j^{-1}(x^{(i)}-\mu_j)\right) \cdot \phi_j}{w_j^{(i)}} \\
&= -\nabla_{\mu_l} \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \frac{1}{2}(x^{(i)}-\mu_j)^T \Sigma_j^{-1}(x^{(i)}-\mu_j) \\
&= \frac{1}{2} \sum_{i=1}^{m} w_l^{(i)} \nabla_{\mu_l} 2\mu_l^T \Sigma_l^{-1} x^{(i)} - \mu_l^T \Sigma_l^{-1} \mu_l \\
&= \sum_{i=1}^{m} w_l^{(i)} \left(\Sigma_l^{-1} x^{(i)} - \Sigma_l^{-1} \mu_l\right)
\end{aligned}$$

Setting this to zero and solving for μ_l therefore yields the update rule

$$\mu_l := \frac{\sum_{i=1}^m w_l^{(i)} x^{(i)}}{\sum_{i=1}^m w_l^{(i)}},$$

which was what we had in the previous set of notes.

Lets do one more example, and derive the M-step update for the parameters ϕ_j. Grouping together only the terms that depend on ϕ_j, we find that we need to maximize

$$\sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j.$$

However, there is an additional constraint that the ϕ_j's sum to 1, since they represent the probabilities $\phi_j = p(z^{(i)} = j; \phi)$. To deal with the constraint that $\sum_{j=1}^k \phi_j = 1$, we construct the Lagrangian

$$\mathcal{L}(\phi) = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \beta(\sum_{j=1}^{k} \phi_j - 1),$$

where β is the Lagrange multiplier.[2] Taking derivatives, we find

$$\frac{\partial}{\partial \phi_j} \mathcal{L}(\phi) = \sum_{i=1}^{m} \frac{w_j^{(i)}}{\phi_j} + 1$$

[2]We don't need to worry about the constraint that $\phi_j \geq 0$, because as we'll shortly see, the solution we'll find from this derivation will automatically satisfy that anyway.

Setting this to zero and solving, we get

$$\phi_j = \frac{\sum_{i=1}^m w_j^{(i)}}{-\beta}$$

I.e., $\phi_j \propto \sum_{i=1}^m w_j^{(i)}$. Using the constraint that $\sum_j \phi_j = 1$, we easily find that $-\beta = \sum_{i=1}^m \sum_{j=1}^k w_j^{(i)} = \sum_{i=1}^m 1 = m$. (This used the fact that $w_j^{(i)} = Q_i(z^{(i)} = j)$, and since probabilities sum to 1, $\sum_j w_j^{(i)} = 1$.) We therefore have our M-step updates for the parameters ϕ_j:

$$\phi_j := \frac{1}{m} \sum_{i=1}^m w_j^{(i)}.$$

The derivation for the M-step updates to Σ_j are also entirely straightforward.

Reinforcement learning

11

11.1 CS221 Lecture notes by Andrew Ng, No 8 and 9

CS221 Lecture notes #8

Reinforcement learning I

In supervised learning, we had a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ for which we had the "right answer" $y^{(i)}$ for every instace $x^{(i)}$. For example, supervised learning algorithms would learn to drive by predicting the actions of an expert human driver. In this set of notes, we'll talk about a different setting called **reinforcement learning**, where we don't know the right answers ahead of time; we only know a "reward function" which tells us the goodness of particular states. (For instance, we might believe that a state where the helicopter is in the air is better than one where it is lying in bits and pieces on the ground.)

In detail, we specify a **reward function** mapping states of the world to real numbers. The algorithm's goal is to find a sequence of actions which maximizes this reward function over time. This temporal component means the algorithm does not simply have to make a one shot decision. For instance, the helicopter must choose actions which not only allow it to stay in the air at this exact moment, but which also keep it stable enough that it can remain in the air continually.

If the world were completely deterministic, it would be easy to maximize such a reward function using techniques from our lectures on search. However, in many robotics systems, the dynamics are **stochastic**, in that the same action doesn't always lead to the exact same result every time. For instance, telling a robot to move one meter forward could typically result in it moving anywhere between 95 and 105 cm forward, due to factors such as slippage of the wheels. Even a small amount of randomness would cause big problems for the deterministic search algorithms we covered earlier in the course.

We will take an approach where we "reward" the robot for desired outcomes and "punish" it for undesired ones. One challenge faced by reinforcement learning is the **credit assignment problem**. Upon reaching a position with negative reward, it may not be obvious which previous action had caused that negative reward. For instance, suppose you are driving and you crash your car. Chances are, you slammed on your breaks shortly before the crash. You wouldn't want to conclude from this that it's a bad idea to ever step on the breaks again. Rather, the crash was probably due to an action you chose much earlier, such as your decision to go 90 MPH down the highway.

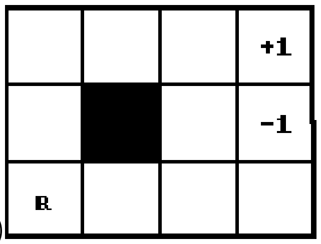
In these notes, we present the standard reinforcement learning formalism, known as the **Markov decision process (MDP).** MDPs allow us to model the (stochastic) dynamics of the world as well as the desired outcomes. As we will see, within this formalism, we can tractably compute the optimal behaviors which maximize the reward function over time.

1 Markov Decision Processes (MDPs)

An MDP is a 5-tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where

- S is the set of states
- A is the set of actions
- P_{sa} gives the **state transition probabilities** for state s and action a. If we are in state s, then for any state s', $P_{sa}(s')$ gives the probability that taking action a will cause us to transition to state s'. Since P_{sa} is a probability distribution, $\sum_{s'} P_{sa}(s') = 1$ and $P_{sa}(s') \geq 0$ for all $s \in S$ and $a \in A$.
- γ is a real-valued **discount factor** in the interval $0 \leq \gamma < 1$ telling us how much we value rewards right now relative to rewards in the future.
- $R : S \mapsto \mathbb{R}$ is the reward function, which measures the desirability of being in each state.

Consider, for instance, the following example from our class textbook, shown in Figure 1 (a). The world is a 4×3 grid, with one obstacle, giving a total of 11 states. There are four possible actions the robot can take, corresponding to moving in each of the four compass directions, labeled $\{N, S, E, W\}$. However, the motion dynamics are noisy, and so if the robot tries to move in a particular direction, there is a 10% chance of it instead moving in the direction 90° to the left, and a 10% chance of it moving in the direction 90° to the right. If the robot's direction of motion causes it to move

(a) 

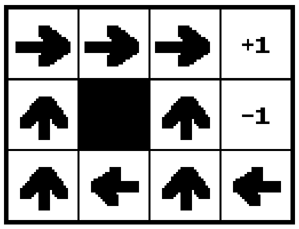
(b) 

Figure 1: (a) An example MDP, taken from Russell & Norvig. The initial location of the robot is given by R. (b) The optimal policy for the MDP, for $\gamma = 0.99$.

into a wall, it simply bounces off and stays put in its current square. For instance, if the robot is in square below the obstacle, trying to move down will result in a 10% chance of moving left, a 10% chance of moving right, and an 80% chance of staying put.

Its goal is to wind up in the upper right corner, so that square is assigned a reward of $+1$. We don't want it to land in the square below, so that square is assigned a reward of -1. As soon as the robot enters one of these two squares, the MDP is over and no more moves are made.[1] We also don't want the robot to dawdle too long, and so all other states are assigned a small negative reward of -0.04 (perhaps corresponding to fuel or battery consumption). Assigning a small negative reward to all states is a common method for preventing a robot from sitting idle.

Let's now define a little more formally how an MDP works. We begin in some state s_0. At each (discrete) time t, we are in some state s_t, we choose some action a_t, and a successor s_{t+1} is drawn according to the transition probabilities, i.e. $s_{t+1} \sim P_{s_t a_t}$. This process generates an infinite sequence of states $s_0, s_1, s_2, \ldots$. The **total payoff** is defined as a weighted sum of the rewards for the states in this sequence:

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots .$$

The contribution of each state to the total payoff is weighted by γ^t, which decreases exponentially quickly with t. Such a weighting is known as **discounting**. In a financial application, the discount factor γ has a natural interpretation as the time value of money. A dollar today is worth more

[1]To treat this more formally, you can imagine that both of these states transition with probability 1 to a "zero-cost absorbing state." This is a state which transitions to itself with probability 1 and has no associated reward.

than a dollar a year from now, because of the interest rate—a dollar placed in a bank will earn you back slightly more than a dollar in a year's time. Small values of γ imply a very fast decay of the value of the rewards per unit time, and hence mean we will be shortsighted. Large values of γ (e.g. very close to 1) mean we will try to maximize our expected payoff long into the future.

We said above that we have to choose an action at each time instant. More specifically, our goal is to find a **policy** π which assigns an action to every state, i.e. $\pi : S \mapsto A$. At each time t, we will choose the action which π assigns the current state, $a_t = \pi(s_t)$. If we take actions in an MDP following the actions specified by a policy π, then we say that we are **executing** policy π in the MDP.

We are interested in computing the optimal policy of the MDP. Before we define the optimal policy, we need some preliminary definitions. First, define the **value function** of a policy π to be a function which takes a state s and returns the expected total payoff if we start at s. More formally, $V^\pi : S \mapsto \mathbb{R}$, where

$$V^\pi(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \;\middle|\; \pi, s_0 = s\right].$$

We can't compute $V^\pi(s)$ directly from this definition, however, because it is a sum of an infinite number of terms. What we will do instead is to define $V^\pi(s)$ recursively in terms of $V^\pi(s')$ for all states s'. This will give a system of linear equations which can be solved. This is called **Bellman's equation for V^π**.

$$\begin{aligned}
V^\pi(s) &= \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \;\middle|\; \pi, s_0 = s\right] \\
&= \mathrm{E}\left[R(s_0) + \gamma\left(R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \cdots\right) \;\middle|\; \pi, s_0 = s\right] \\
&= R(s) + \gamma \mathrm{E}\left[R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \cdots \;\middle|\; \pi, s_0 = s\right] \\
&= R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s')\ \mathrm{E}\left[R(s_1) + \gamma R(s_2) + \gamma R(s_3) + \cdots \;\middle|\; \pi, s_1 = s'\right] \\
&= R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s')\ V^\pi(s') \qquad (1)
\end{aligned}$$

This is a system of linear equations, where the variables are the values of $V^\pi(s)$ for the different states s. If n is the number of states, there are n equations with n unknowns. Such a system can be solved in closed form using standard techniques. In the equation above, $R(s)$ is sometimes also called the **immediate reward**.

Now, for a given state s, we can define the **optimal value** of s, denoted $V^\star(s)$, as the largest expected total payoff starting from s which can be achieved by any policy:

$$V^\star(s) = \max_\pi V^\pi(s).$$

We don't want to compute $V^\star(s)$ directly from this definition (i.e., by enumerating all policies). Instead, we will make use of a version of Bellman's equation for $V^\star$:

$$V^\star(s) = R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V^\star(s'). \qquad (2)$$

You should convince yourself that Bellman's equation must hold true for the optimal values $V^\star$. The converse is more subtle. You'll have a chance to prove, in Problem Set 2, that $V^\star$ is uniquely defined by Bellman's equation.

Now, rather than define the optimal policy directly, we define a particular policy $\pi^\star$, which will turn out to be the optimal policy. Define $\pi^\star$ to be the policy which looks the best according to $V^\star$, i.e.

$$\pi^\star(s) = \arg\max_a \sum_{s'} P_{sa}(s')V^\star(s').$$

It's a fact, which we won't prove, that

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

(Make sure you understand all the notation used here.) In other words, $\pi^\star$ is the **optimal policy**, or the one which achieves the highest expected total payoff for each state.

Figure 1 (b) shows the optimal policy for the grid world.

2 Solving MDPs

We have just defined our goal as finding the optimal policy $\pi^\star$, and now we introduce algorithms to do that.

2.1 Value iteration

We gave a formula for computing the optimal policy $\pi^\star$ in terms of the optimal value function $V^\star$, so one way to proceed is to compute the optimal value $V^\star(s)$ for each state s, and then use $V^\star$ to get $\pi^\star$. **Value iteration** is an iterative algorithm which essentially changes the equality in Bellman's equation (Equation 2) into an update rule.

Specifically, value iteration works as follows:

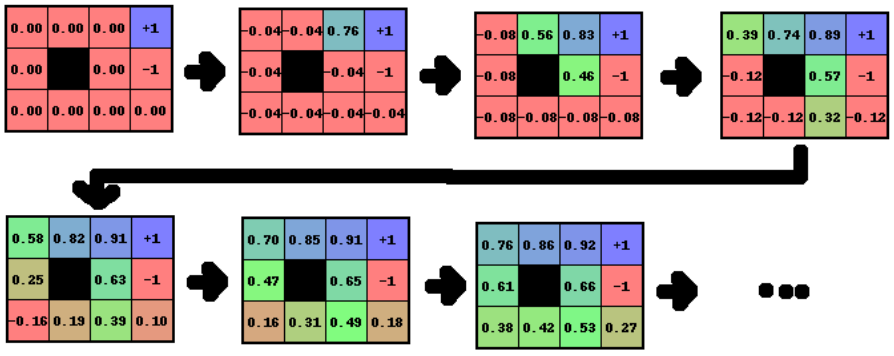


Figure 2: An example of value iteration applied to our MDP, for $\gamma = 0.999$.

```
Initialize $V(s) = 0$ for all states $s$.

Repeat until convergence:

    For every state $s \in S$, update
        $V(s) := R(s) + \max_a \gamma \sum_{s' \in S} P_{sa}(s')V(s')$
```

This procedure will gradually cause $V(s)$ to converge to the optimal value function $V^\star(s)$. An example is shown in Figure 2.

Notice that our above definition is ambiguous. Suppose it's time to update a state s on the i^{th} pass through all of the states, and for some other state s', $V(s')$ has already been updated on the i^{th} pass. Which value of $V(s')$ do we use: the one from the $i-1^{st}$ pass, or the one which was newly assigned on the i^{th} pass? It turns out that both versions give a correct algorithm. Using the value from the $i-1^{st}$ pass is known as **synchronous** updates, while using the value from the i^{th} pass is known as **asynchronous** updates.

Somtimes it's convenient to think of V a vector, where the j^{th} component of V corresponds to $V(s_j)$. In the synchronous updates version of value iteration, we sometimes refer to one pass through all of the states as the **Bellman operator** B. In this notation, each pass through the states can be written as $V := B(V)$.

2.2 Policy iteration

Policy iteration is another algorithm based on a similar intuition. Policy iteration also uses the Bellman equations as update rules in an iterative

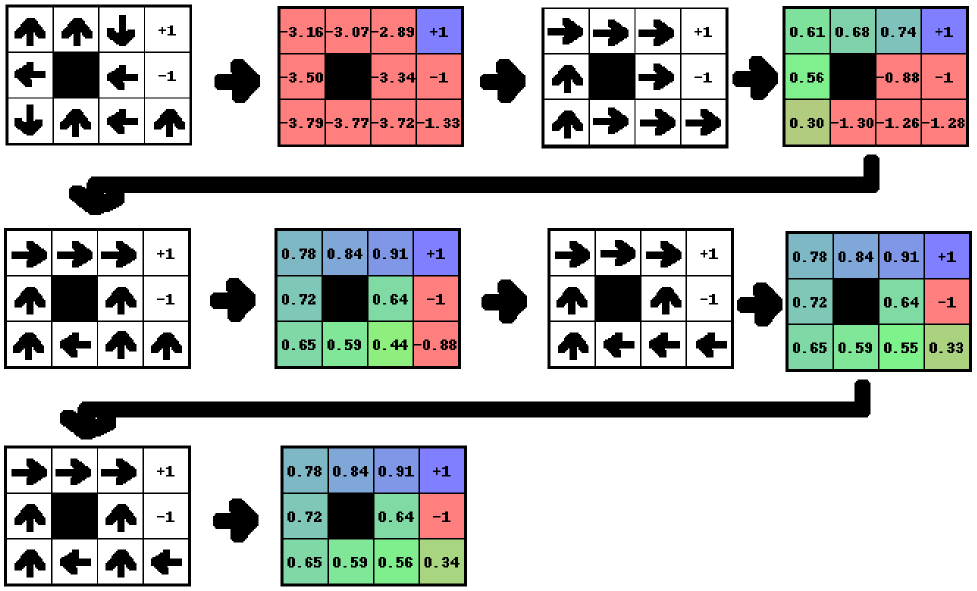


Figure 3: An example of policy iteration applied to our MDP, for $\gamma = 0.99$.

algorithm, but does so in a slightly different way. In policy iteration, we alternate between computing the expected total payoff function V^π for a given policy and choosing a new policy π based on our current estimate of the value of each state. Our estimate $V(s')$ is taken to be the *actual* expected total payoff for state s' under the policy π, as computed by solving the system of linear equations.

Policy iteration can be written as follows:

```
Initialize $\pi$ randomly.

Repeat until convergence:

    Let $V := V^\pi$. (In other words, compute $V^\pi$ from (1).)
    Let $\pi(s) := \arg\max_a \sum_{s' \in S} P_{sa}(s')V(s')$
```

In a finite number of iterations, V will converge to $V^\star$, and π will converge to $\pi^\star$. An example is shown in Figure 3.

What are the advantages and disadvantages of policy iteration relative to value iteration? The advantage is that, rather than using its previous (possibly very inaccurate) estimate of the value of a particular state s, it actually computes the exact value of the state relative to some policy π. Hence, if the current policy is somewhat similar to the optimal one, then V should be very accurate. In practice, policy iteration takes many fewer iterations to converge than value iteration.[2] On the other hand, it's much more expensive to compute one iteration of policy iteration than one iteration of value iteration. For each update, value iteration only requires taking a maximum of a set of numbers, while policy iteration requires solving a set of linear equations. Therefore, policy iteration is perhaps most effective in domains with a small number of states, and value iteration for larger problems.

[2]It is still an open problem to find good bounds for the number of iterations required for policy iteration to converge. The best known bound is exponential in the number of states, but there are no known examples which actually take exponentially long.

CS221 Lecture notes #9

Reinforcement learning II

So far, we have considered reinforcement learning using the framework of discrete MDPs. In particular, we assumed a finite state space. We now discuss ways to handle problems where the state space is continuous. Specifically, we discuss two general methods for solving continuous MDPs. The first, based on discretization, will approximate the continuous value function in terms of a value function defined on a discrete set of points drawn from the continuous space. The second, called fitted value iteration, will compute an approximation to the value function using an iterative learning algorithm. Fitted value iteration will allow us to scale MDP algorithms to much larger numbers of dimensions than would be possible using discretization.

1 Discretization

Suppose you want to apply reinforcement learning to driving a car. Let's say we model the state of the car as the vector

$$s_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \in \mathbb{R}^3,$$

where x_t and y_t give the location of the car at time t, and θ gives its orientation. Assume for now that we have a finite set of actions A (e.g. turn left, step on the breaks, etc.).[1] We suppose we have a simulator which takes a state-action pair (s_t, a_t) for time t and returns a state s_{t+1} for time $t+1$. In

[1]In some problems, we can have continuous actions also. For instance, when driving, we can control how much to turn left, how strongly to step on the brakes, and so on. But for most problems, A has only a small number of degrees of freedom, and so it's not hard to discretize. For instance, we might realistically model the state of a car as a vector

other words, it will sample s_{t+1} from the distribution $P_{s_t a_t}$. We treat the simulator as a black box which takes a state-action pair and returns a successor state:

s_t
a_t
Simulator
s_{t+1}

In some cases, we have a good model of the physics, and therefore we can determine the transition probabilities $P_{s_t a_t}$ through physical simulation. Often, however, it's hard to construct a good physical model a priori. In these cases, the simulator itself has to be learned.

Sometimes, the simulator is **deterministic**, in that it will always return the same state s_{t+1} for a particular state-action pair (s_t, a_t). Sometimes, the simulator is **stochastic**, where s_{t+1} is a random function of s_t and a_t.

Say we have a continuous state space S. One way to apply the techniques from the previous set of notes is to **discretize** S to obtain a discrete set of states $\bar{S} = \{\bar{s}^{(1)}, \bar{s}^{(2)}, \ldots, \bar{s}^{(n)}\}$. For instance, we might choose to break up the continuous state space S into boxes and let $\bar{S}$ have one discrete state $\bar{s}^{(i)}$ for each of these boxes. We will refer to this method as **simple grid discretization**. This is not necessarily the best approach for many problems, but it is simple to implement, and is often good enough. In these notes, s will always denote a state in the original continous state, and $\bar{s}$ will denote one of the discrete states.

After discretizing our continuous space, we typically need to estimate the transition probabilities $P_{\bar{s}a}(\bar{s}')$. We do this by taking a lot of samples and estimating $P_{\bar{s}a}(\bar{s}')$ as the proportion of times we landed in discrete state $\bar{s}'$ after taking action a in some continuous state associated with discrete state $\bar{s}$. More formally, we use the algorithm shown in Figure 1.

We also need to estimate the reward function for our discretized MDP. This can be done analogously to how we estimate the transition probabilities. Specifically, we take a large number of samples from S, and then for each discrete state $\bar{s}$, we take $R(\bar{s})$ to be the mean value of $R(s)$ for all of our samples s which were associated with $\bar{s}$.

$s \in \mathbb{R}^6 = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ giving the location and orientation, and the rate of change of each. The set of actions, however, can be given as a vector in $\mathbb{R}^2$. Therefore, we will focus our attention on dealing with continuous state spaces, and assume the set of actions is discrete.

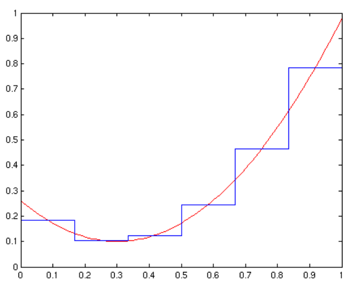
```
For $i = 1$ to $k$,
    Sample $s_t$ randomly from the continuous state space $S$.
    For each action $a \in A$,
        "Try" action $a$ from state $s_t$. In other words, let $s_{t+1}$ be the
        state returned by the simulator when given state $s_t$ and action $a_t$.
        Find the discrete states $\bar{s}_t$ and $\bar{s}_{t+1}$ associated with the
        continuous states $s_t$ and $s_{t+1}$.
Estimate
```

$$P_{\bar{s}_t a}(\bar{s}_{t+1}) = \frac{\#\text{ times action } a \text{ in state } \bar{s}_t \text{ caused a transition to state } \bar{s}_{t+1}}{\#\text{ times we tried action } a \text{ in state } \bar{s}_t}.$$

Figure 1: An algorithm for estimating the transition probabilities in a discretized space.

This process gives us the complete specification of a discrete MDP, which we can then solve using the techniques such as value iteration of policy iteration. This will give us the optimal value function $V^\star(\bar{s})$ and the optimal policy $\pi^\star(\bar{s})$ for the discrete problem. How do we use this to choose a policy for the original continuous problem? If we want to choose an action a_t for a given continuous state s_t, we can map s_t to the corresponding discrete state $\bar{s}_t$ and choose the action $a_t = \pi^\star(\bar{s}_t)$.

Formulating a problem as a continuous MDP has several advantages over the motion planning algorithms presented earlier in this course. Specifically, it can handle cases that deterministic motion planning methods can't, including *nonholonomic motion* and *stochasticity*. As a consequence of its being able to handle nonholonomic problems, it can also account for the **dynamics** of the problem. In other words, it can account for the rates of change of the different variables. For instance, in the notes on motion planning, we represented the configuration of a helicopter as a vector in $\mathbb{R}^6$, with three dimensions for the location of the helicopter and three dimensions for its orientation. In the MDP framework, we can represent the state as a vector in $\mathbb{R}^{12}$ which also includes the rate of change of each of these variables. This means a state where the helicopter is flying steadily in the air will be treated as distinct from one where it is plummeting towards the ground.

(a) 

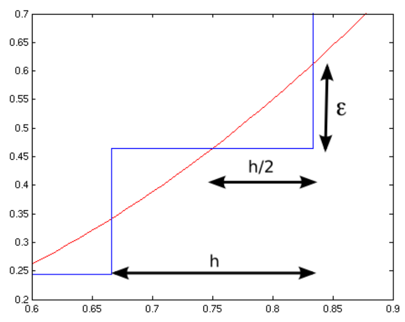
(b) 

Figure 2: (a) An example of simple grid discretization applied to a real-valued function on the interval $[0, 1]$. (b) A close-up with the edge width h and error ε labeled.

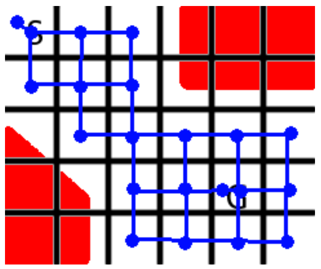
2 Linear interpolation

Previously, we introduced a simple algorithm for discretizing a continuous state space. We supposed we had a function which would assign each continuous state s to some discrete state $\bar{s}$. The value we assign to s would simply be the value of $\bar{s}$. For the moment, let's restrict ourselves to one-dimensional state spaces, and consider the more general problem of discretizing some real-valued function $f : \mathbb{R} \mapsto \mathbb{R}$ on the interval $[0, 1]$. What does simple discretization look like when applied to this problem? We would partition $[0, 1]$ into some discrete set of intervals. We then approximate the value of f in these intervals as being constant, to get an approximation $\bar{f}$ that is **piecewise constant**. An example is shown in Figure 2a. In the sequel, we will use h to denote the width of each of the intervals. (E.g., if you discretize $[0, 1]$ into n discrete intervals, then we would have $h = 1/n$.)

How good is this approximation? Clearly, this will depend how finely we discretize the input domain. I.e., it will depend on how small h is. We are specifically interested in the problem of how well $\bar{f}$ approximates f as $h \rightarrow 0$ (i.e., in the limit of finer and finer discretizations). Figure 2b shows a close-up view of f and its approximation $\bar{f}$. If the function f has approximately gradient m in this region, then it is possible to show that the maximum error of the approximation is given by

$$\varepsilon \approx \frac{h}{2}m = O(mh).$$

This big-O notation is in the limit of $h \rightarrow 0$. The error, therefore, decreases

(a) 

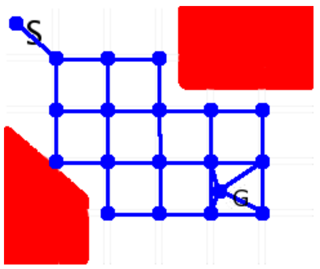
(b) 

Figure 3: An example of the difference between the two kinds of grid discretization, using a configuration space from our motion planning lecture. (a) The kind of discretization where the space is divided into "boxes." (b) The kind of discretization where we lay down a lattice of grid points in the state space. This is the kind of discretization we will be using.

linearly with the density at which the points are sampled.[2] If you want your answer to be 10 times as accurate, you need to sample 10 times as many points. This problem gets worse when we add more dimensions; in three dimensions, sampling 10 times more finely requires 10^3 times as many samples. Hence, we would need to choose 1000 times as many points to get one more significant digit accuracy.

Fortunately, we can do better than this with relatively little additional computational cost. Rather than approximate f with a piecewise constant function, we will use a **piecewise linear** approximation. To explain this, we will need to take a different view of discretization. Previously, we talked about discretization in terms of splitting up the state space into a number of "boxes." We then associated each of the "boxes" with a value. This is illustrated in Figure 3a. However, we will now take a different view of discretization in which we lay down a lattice of grid points in the state space. We will then associate each of the **lattice points** with a value. This is shown in Figure 3b. Sometimes, the "boxes" view leads to much more natural algorithms, and sometimes the "lattice" view does. If you're ever choosing some discretization for a problem, remember this picture, and make sure to choose the most appropriate one for your problem.

In detail, lets again consider the case of discretizing a function f over

[2]You are probably used to big-O notation where $n \to \infty$. Here, the big-O notation represents the limit as $h \to 0$. More formally, $g_1(h) = O(g_2(h))$ if for any constant $c > 0$, there exists some constant d such that $g_1(h) \le c g_2(h)$ whenver $0 < h < d$.

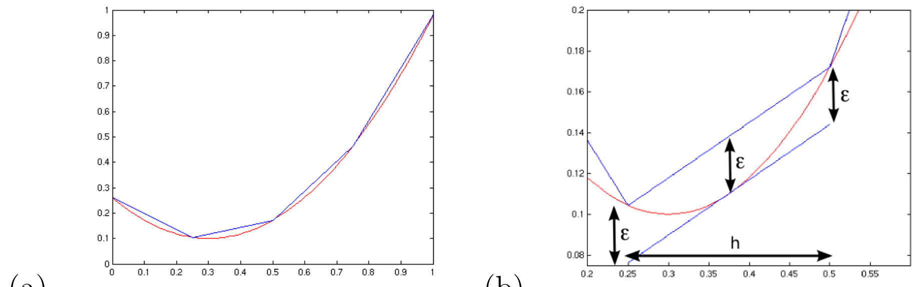
(a) (b)


Figure 4: (a) An example of linear interpolation applied to a real-valued function on the interval $[0, 1]$. (b) A close-up with the edge width h and error ε labeled.

$[0, 1]$. Suppose we choose $n + 1$ lattice points, at locations

$$\bar{x}^{(0)} = \frac{0}{n}, \bar{x}^{(1)} = \frac{1}{n}, \ldots, \bar{x}^{(n)} = \frac{n}{n}.$$

If we are trying to approximate $f(x)$ for some $x \in \mathbb{R}$, let $x^{(-)}$ denote the nearest lattice point to the left of x, and $x^{(+)}$ the nearest one to the right of x. We then **interpolate** between the values $f(x^{(-)})$ and $f(x^{(+)})$. More formally, let $\alpha = \frac{x - x^{(-)}}{x^{(+)} - x^{(-)}}$ denote the fraction of the space between $x^{(-)}$ and $x^{(+)}$ which lies to the left of x. Then we approximate

$$\bar{f}(x) = (1 - \alpha) f(x^{(-)}) + \alpha f(x^{(+)}).$$

This process is known as **linear interpolation**, and it is illustrated in Figure 4a. Visually, it certainly looks like a better appproximation than our piecewise constant one.

How accurate is linear interpolation? As before, we check how fast ε decreases as $h \to 0$. It turns out that the error decreases quadratically with h, i.e., $\varepsilon = O(h^2)$. The precise statement of this result is somewhat technical, but it suffices to say linear interpolation gives significantly better results than piecewise constant approximations. Put the other way, with linear interpolation, the number of grid points we need is roughly the *square root* of what we would need with piecewise constant approximations.

2.1 Value iteration updates

Still limiting ourselves to the 1-dimensional case, we're going to show how to apply value iteration when we use linear interpolation on our discretized points.

Suppose our state space S is an interval in $\mathbb{R}$, and we have discretized it into a grid of n points $\bar{s}^{(1)}, \ldots, \bar{s}^{(n)}$. We will explicitly determine the value function $V(\bar{s})$ at the grid points, and then compute $V(s)$ with linear interpolation everywhere else. If s is between $\bar{s}^{(i)}$ and $\bar{s}^{(i+1)}$, and $\alpha = \frac{s-\bar{s}^{(i)}}{\bar{s}^{(i+1)}-\bar{s}^{(i)}}$, we define

$$V(s) = (1-\alpha)V(\bar{s}^{(i)}) + \alpha V(\bar{s}^{(i+1)}).$$

We will approximately solve for $V(\bar{s}^{(i)})$ for each lattice point $\bar{s}^{(i)}$ using value iteration. Assume for simplicity that our simulator is deterministic, i.e., there is some function $\delta : S \times A \mapsto S$ which gives us the resulting state s' whenever we feed a state-action pair (s, a) into the simulator. In this case, we do the following:

```
Initialize $V(\bar{s}^{(i)}) = 0$ for all grid points $\bar{s}^{(i)}$.

Repeat until convergence:

    For each grid point $\bar{s}^{(i)}$:
        For each action $a \in A$:
            Let $s'_a = \delta(\bar{s}^{(i)}, a)$.
            Compute $V(s'_a)$ via linear interpolation.
        Set $V(\bar{s}^{(i)}) := R(\bar{s}^{(i)}) + \gamma \max_a V(s'_a)$.
```

As with ordinary value iteration, this will converge, giving us the (approximate) optimal policy $V^\star$. We can choose our policy as:

$$\pi^\star(s) = \arg\max_a V^\star(\delta(s, a)).$$

Essentially, we are using our simulator to do one-step lookahead to see which action takes us to the state with the highest value.

We just assumed the simulator was deterministic. If the simulator is stochastic, then rather than choosing a single point $s' = \delta(\bar{s}^{(i)}, a)$, we sample k successor states, and use the average value of these successor states in the update rule. Finally, to choose an action a_t for a given continuous state s_t, we run the simulator k times for each action a to get k points $s'_{a1}, \ldots, s'_{ak}$. We choose:

$$\pi^\star(s) = \arg\max_a \frac{1}{k} \sum_{i=1}^{k} V^\star(s'_{ai}).$$

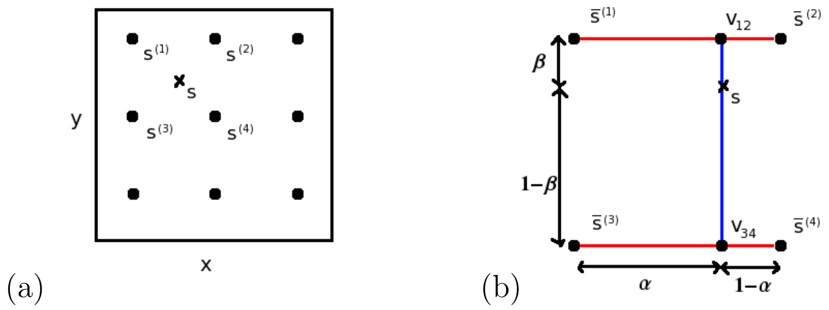


Figure 5: (a) An example of a discretized MDP. We are trying to estimate the value of state s based on the values assigned to our lattice points $\bar{s}^{(1)}, \ldots, \bar{s}^{(4)}$. (b) How to apply bilinear interpolation to this MDP. First, the value function is interpolated between $\bar{s}^{(1)}$ and $\bar{s}^{(2)}$ to get V_{12}, and between $\bar{s}^{(3)}$ and $\bar{s}^{(4)}$ to get V_{34}. Then we interpolate between V_{12} and V_{34} to get $V(s)$.

3 Multilinear interpolation

We now describe the generalization of linear interpolation to higher dimensional state spaces. Suppose now that we have a two-dimensional continuous state space S, which we have discretized into a grid as shown in Figure 5a. Assume we somehow have a value function V defined on all of the grid points. How do we compute $V(s)$? Notice first that we can't simply use piecewise linear interpolation. The values of V at our four grid points $\bar{s}^{(1)}, \ldots, \bar{s}^{(4)}$ have to be specified with four real numbers. On the other hand, if we tried to predict $V(s)$ with a linear function, e.g.

$$V(s) = \theta_0 + \theta_1 s_1 + \theta_2 s_2,$$

we only have three degrees of freedom to specify V. Hence, we can't always come up with a linear function which matches V at each of the four grid points $\bar{s}^{(1)}, \ldots, \bar{s}^{(4)}$.

Instead, we will use **bilinear interpolation**, as demonstrated in Figures 5b and 6. First, we interpolate linearly between $\bar{s}^{(1)}$ and $\bar{s}^{(2)}$ to get V_{12}. Similarly, we compute V_{34} by interpolating $\bar{s}^{(3)}$ and $\bar{s}^{(4)}$. Finally, we interpolate between V_{12} and V_{34} to get $V(s)$.

It might seem funny to interpolate in the x direction before interpolating in the y direction. Does this give preference to one coordinate over the other? Actually, it turns out that we get the same result no matter which coordinate we interpolate first. We can show this by taking our definition of bilinear

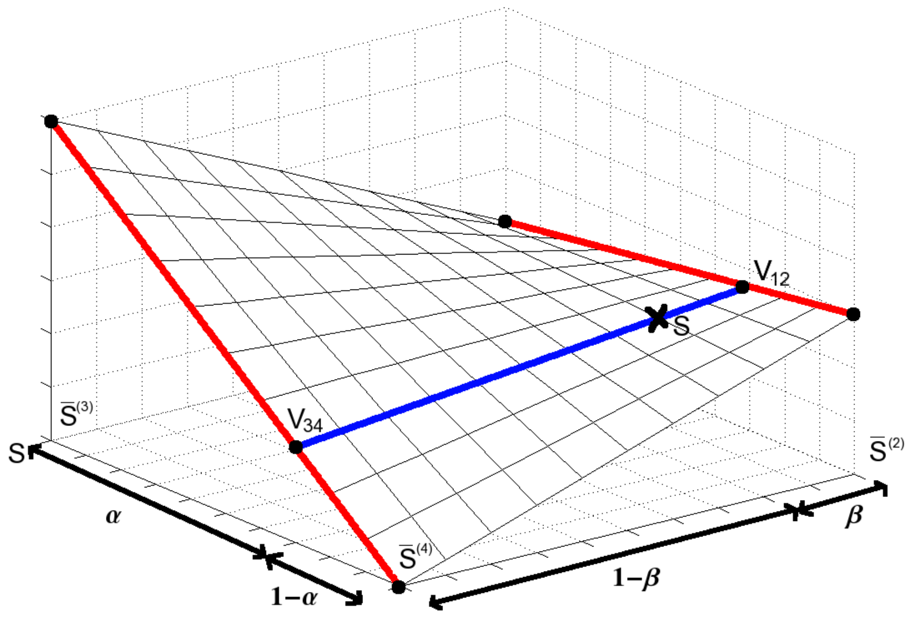


Figure 6: An example of bilinear interpolation of the value function between four lattice points $\bar{s}^{(1)}, \ldots, \bar{s}^{(4)}$.

interpolation and expanding out the polynomial in terms of α and β:

$$\begin{aligned} V(s) &= (1-\beta)\left((1-\alpha)V(\bar{s}^{(1)}) + \alpha V(\bar{s}^{(2)})\right) \\ &\quad +\beta\left((1-\alpha)V(\bar{s}^{(3)}) + \alpha V(\bar{s}^{(4)})\right) \\ &= (1-\alpha)(1-\beta)V(\bar{s}^{(1)}) + \alpha(1-\beta)V(\bar{s}^{(2)}) \\ &\quad +(1-\alpha)\beta V(\bar{s}^{(3)}) + \alpha\beta V(\bar{s}^{(4)}). \end{aligned} \tag{1}$$

In other words, the weights of the points to the left of s are proportional to $1-\alpha$, while the weights of the points to the right of s are proportional to α. The same is true for the vertical direction. Clearly, if we had begun by interpolating vertically rather than horizontally, we would have arrived at the same formula.

Let's stop to think briefly about what it means for a function to be bilinear. Our function $V(s)$, as computed by the polynomial (1), is linear in each coordinate of s taken individually. In other words, as we move s directly north, our value function changes linearly. However, $V(s)$ is *not* linear in s. As s moves from one corner $\bar{s}^{(3)}$ to the opposite corner $\bar{s}^{(2)}$, $V(s)$ clearly changes nonlinearly. A function which is linear in each of its inputs taken separately is called bilinear.

Our example was in two dimensions, but if we apply the exact same technique in higher dimensional spaces, we get **multilinear interpolation**.

4 Fitted value iteration

Unfortunately, both of the methods we described above (simple grid discretization and multilinear interpolation) suffer from exponential growth in the problem size as the number of dimensions is increased. (We saw exactly the same problem with the discretization techniques we introduced in our motion planning lecture.) In reinforcement learning, this exponential growth is often referred to as the **curse of dimensionality**. Simple grid discretization works well in 3 dimensions, is sometimes OK in 4 dimensions, and occasionally works in 6 dimensions with much luck and effort. With multilinear interpolation, we can buy ourselves a couple extra dimensions; it sometimes works in 6 dimensions, but only rarely in 7 or 8.

But in our supervised learning lectures, we used algorithms which performed well for hundreds or thousands of dimensions. Linear regression was able to handle large numbers of dimensions because its hypotheses were restricted to be linear functions of the inputs. This suggests learning a value function which is a linear function of some predefined set of features. More formally, suppose we have a **feature map** $\phi : S \mapsto \mathbb{R}^n$, which associates with each state s a **feature vector** $\phi(s) \in \mathbb{R}^n$. For instance, if our state space is one dimensional, and we want to approximate our value function as a cubic polynomial, we might use the feature vector

$$\phi(s) = \begin{bmatrix} s \\ s^2 \\ s^3 \end{bmatrix}.$$

Or, if our state space is n-dimensional, we might simply choose as our feature map the identity function $\phi(s) = s$.

We will use the **value function approximation**, and approximate the value function V as a linear function of the feature vector $\phi(s)$, i.e.

$$V(s) = V_\theta(s) = \theta^T \phi(s) = \sum_{i=1}^{n} \theta_i \phi_i(s).$$

Note that this is completely analogous to linear regression, where our hypotheses h_θ were linear functions of the input variables x_i:

$$h_\theta(x) = \theta^T x = \sum_{i=1}^{n} \theta_i x_i.$$

We can learn the weights θ using another variant of value iteration called **fitted value iteration**. If our simulator is deterministic, the algorithm is as shown in Figure 7.

Eventually, we will find a linear approximation V_θ to the optimal value function. Unlike the previous algorithms we presented, fitted value iteration is not guaranteed to converge, but in practice, it will usually converge or approximately converge. As with multilinear interpolation, we can then find the optimal policy with

$$\pi^\star_\theta(s) = \arg\max_a V^\star_\theta(\delta(s, a)).$$

If the simulator is stochastic, then when setting $y^{(i)}$, we must estimate the expected value of the successor state by sampling successor states from the simulator and taking the average. Given a continuous state s, for each action a, we run the simulator k times for each action a to get k points $s'_{a1}, \ldots, s'_{ak}$. We choose:

$$\pi^\star(s) = \arg\max_a \frac{1}{k} \sum_{i=1}^{k} V^\star(s'_{ai}).$$

```
Initialize $\theta = 0$.

Sample a set of states $\bar{s}^{(1)}, \bar{s}^{(2)}, \ldots, \bar{s}^{(m)}$ randomly from $S$.

Repeat until convergence:

    For $i = 1$ to $m$:
        Set $x^{(i)} = \phi(\bar{s}^{(i)})$.
        Set
            $$\begin{aligned} y^{(i)} &= R(\bar{s}^{(i)}) + \max_a V(s_a^{(i)}) \\ &= R(\bar{s}^{(i)}) + \max_a \theta^T \phi(s_a^{(i)}), \end{aligned}$$
        where $s_a^{(i)} = \delta(\bar{s}^{(i)}, a)$.
    Choose
        $$\begin{aligned} \theta &:= \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} (V_\theta(\bar{s}^{(i)}) - y^{(i)})^2 \\ &= \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2 \end{aligned}$$
```

Figure 7: Fitted value iteration algorithm.

Part III

Probabilistic Reasoning

Bayesian networks and Markov models

12

12.1 CS221 Lecture notes by Andrew Ng, No 11–13

CS221 Lecture notes #11

Bayesian networks

1 Probability Review

This material on Bayesian networks (Bayes nets) will rely heavily on several concepts from probability theory, and here we give a very brief review of these concepts. For more complete coverage, see Chapter 13 of the class textbook.

Although the Bayes net framework can accommodate continuous random variables, we will limit ourselves to discrete random variables X which can take on a finite set of values $x^1, \ldots, x^d$. In general, we will use uppercase letters to denote random variables and lowercase letters to denote the values those variables may take on. The probability that X takes the value x will be denoted $\mathrm{P}(X = x)$, or when there is no risk of ambiguity, $\mathrm{P}(x)$. The **joint distribution** over n random variables $X_1, \ldots, X_n$ encodes the probability of a particular assignment to all of the variables, i.e. $\mathrm{P}(X_1 = x_1, \ldots, X_n = x_n)$, or simply $\mathrm{P}(x_1, \ldots, x_n)$.

The **conditional probability** that a random variable X takes on the value x given some other random variable Y takes on the value y is written $\mathrm{P}(x \mid y)$, and is defined as:

$$\mathrm{P}(x \mid y) = \frac{\mathrm{P}(x, y)}{\mathrm{P}(y)}.$$

More generally, for a set of random variables $X_1, \ldots, X_m$ and $Y_1, \ldots, Y_n$, we can write:

$$\mathrm{P}(x_1, \ldots, x_m \mid y_1, \ldots, y_n) = \frac{\mathrm{P}(x_1, \ldots, x_m, y_1, \ldots, y_n)}{\mathrm{P}(y_1, \ldots, y_n)}.$$

We will use bold letters to denote sets of random variables and the values they might take. If $\mathbf{X} = \{X_1, \ldots, X_m\}$ and $\mathbf{Y} = \{Y_1, \ldots, Y_n\}$, we can rewrite this definition as:

$$\mathrm{P}(\mathbf{x} \mid \mathbf{y}) = \frac{\mathrm{P}(\mathbf{x}, \mathbf{y})}{\mathrm{P}(\mathbf{y})}.$$

We refer to the quantity $\mathrm{P}(\mathbf{Y} = \mathbf{y})$ as the **marginal probability** of $\mathbf{Y}$ when we want to emphasize that we are ignoring $\mathbf{X}$. If we are given the joint distribution over two sets of random variables $\mathbf{X}$ and $\mathbf{Y}$, and $\mathbf{x}$ and $\mathbf{y}$ denote joint assignments to $\mathbf{X}$ and $\mathbf{Y}$, we can retrieve the marginal probability of $\mathbf{Y}$ by **marginalizing** over all of the possible assignments to $\mathbf{X}$, i.e.,

$$\mathrm{P}(\mathbf{y}) = \sum_{\mathbf{x}} \mathrm{P}(\mathbf{x}, \mathbf{y}).$$

By plugging this equation into our definition of conditional probability, we get **Bayes' Rule**:

$$\mathrm{P}(\mathbf{x} \mid \mathbf{y}) = \frac{\mathrm{P}(\mathbf{x}, \mathbf{y})}{\sum_{\mathbf{x}'} \mathrm{P}(\mathbf{x}', \mathbf{y})}.$$

Two sets of random variables $\mathbf{X}$ and $\mathbf{Y}$ are **independent** if

$$\mathrm{P}(\mathbf{x}, \mathbf{y}) = \mathrm{P}(\mathbf{x})\mathrm{P}(\mathbf{y}).$$

By dividing both sides through by $\mathrm{P}(\mathbf{y})$, we see that this definition is equivalent to

$$\mathrm{P}(\mathbf{x} \mid \mathbf{y}) = \mathrm{P}(\mathbf{x}).$$

More generally, we say two sets of random variables $\mathbf{X}$ and $\mathbf{Y}$ are **conditionally independent** given a third set of random variables $\mathbf{Z}$ if

$$\mathrm{P}(\mathbf{x}, \mathbf{y} \mid \mathbf{z}) = \mathrm{P}(\mathbf{x} \mid \mathbf{z})\mathrm{P}(\mathbf{y} \mid \mathbf{z}),$$

or equivalently,

$$\mathrm{P}(\mathbf{x} \mid \mathbf{y}, \mathbf{z}) = \mathrm{P}(\mathbf{x} \mid \mathbf{z}).$$

Finally, it is worth noting the **chain rule** for joint probabilities. If $\mathbf{X}$ and $\mathbf{Y}$ are two sets of random variables, we can simply multiply both sides by $\mathrm{P}(\mathbf{y})$ in the definition of conditional probability to find that

$$\mathrm{P}(\mathbf{x}, \mathbf{y}) = \mathrm{P}(\mathbf{x} \mid \mathbf{y})\mathrm{P}(\mathbf{y}).$$

If we apply this formula repeatedly, we find that for any sets of random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$,

$$\mathrm{P}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \prod_{i=1}^{n} \mathrm{P}(\mathbf{x}_i \mid \mathbf{x}_1, \ldots, \mathbf{x}_{i-1}).$$

2 Motivation

When we studied constraint satisfaction problems and propositional satisfiability, we assumed we had a set of "hard" constraints on the world. The CSP formalism only distinguished between those assignments to variables which were possible and those which were impossible. However, we're often interested in cases where many assignments are possible, but only a subset of them are likely. For instance, imagine you are a doctor, and a patient comes to you with a cough, a sneeze, and red eyes. One possibility is that the patient has the flu; this would account for all three symptoms. Another possibility is that the patient simultaneously has the flu (which accounts for the sneezing), lung cancer (which accounts for the cough), and a corneal ulcer (which accounts for the red eyes). Strictly speaking, both of these possibilities are consistent with our observations of the patient's symptoms. The latter situation is clearly unlikely, however, and we want to be able to treat the two differently. In order to do this, we need a way to encode uncertainty.

We will do this by giving a probability distribution over all possible states of the world. In our medical diagnosis problem, suppose we represent the state of the world with three binary random variables: flu (F), allergy (A), and sinus (S). We say F takes the value f if the patient has the flu, and it takes the value $\bar{f}$ otherwise. We can represent the **joint distribution** over these three variables with a table of 8 numbers, each one representing the probability of a certain assignment to the three variables. For now, we assume no restrictions on the joint distribution other than that the probabilities sum to 1.

F	A	S	
f	a	s	0.027
f	a	$\bar{s}$	0.003
f	$\bar{a}$	s	0.162
f	$\bar{a}$	$\bar{s}$	0.108
$\bar{f}$	a	s	0.014
$\bar{f}$	a	$\bar{s}$	0.056
$\bar{f}$	$\bar{a}$	s	0.0063
$\bar{f}$	$\bar{a}$	$\bar{s}$	0.6237

With a joint distribution such as the one given above, we can answer any **query** about the domain. More specifically, we can answer any question about the probability of a particular assignment to one set of variables, possibly conditioned on the values of other variables. For example, suppose we're interested in the probability that the patient has the flu ($F = f$) given that he has sinus trouble ($S = s$). To do this, we apply Bayes' Rule:

$$\mathrm{P}(f \mid s) = \frac{\mathrm{P}(f, s)}{\mathrm{P}(f, s) + \mathrm{P}(\bar{f}, s)}.$$

Note that we use $\mathrm{P}(f \mid s)$ as a shorthand for $\mathrm{P}(F = f \mid S = s)$, the probability that F takes the value f given that S takes the value s. To find $\mathrm{P}(f, s)$, we add up the entries from the table which are consistent with that assignment, i.e.

$$\mathrm{P}(f, s) = 0.027 + 0.162 = 0.184.$$

Similarly,

$$\mathrm{P}(\bar{f}, s) = 0.014 + 0.0063 = 0.0203.$$

By plugging these numbers into Bayes' Rule, we get:

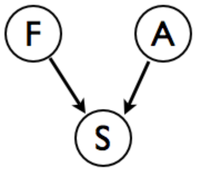
$$\mathrm{P}(f \mid s) = \frac{0.184}{0.184 + 0.0203} = 0.903.$$

Therefore, we can conclude that the probability that the patient has the flu is about 90%.

Explicitly specifying the joint assignment in a table works for tiny examples such as the one above, but it doesn't scale because the size of the representation grows exponentially in the number of variables. Not only does this make actually computing the answers to queries very difficult, but a full joint distribution is unintuitive and hard for humans to specify exactly. Bayesian networks provide a compact and more computationally tractable way to represent joint distributions.

3 Bayesian network definition

A **Bayesian network (Bayes net)** is a directed acyclic graph, where nodes correspond to random variables and edges correspond to direct influence of one variable on another. Each node is associated with a **conditional probability table (CPT)** which gives the probability that the corresponding variable takes on a particular value given the values of its parents. For instance, we might use the following network to represent our medical diagnosis example:



Intuitively, this encodes the information that sinus problems depend directly on having a flu or allergies. The CPTs might be:

P(F):

f	$\bar{f}$
0.3	0.7

P(A):

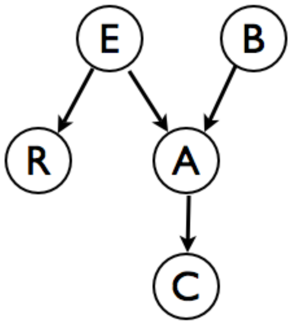
a	$\bar{a}$
0.1	0.9

P($S \mid F, A$):

		s	$\bar{s}$
f	a	0.9	0.1
f	$\bar{a}$	0.6	0.4
$\bar{f}$	a	0.2	0.8
$\bar{f}$	$\bar{a}$	0.01	0.99

For example, this indicates that the probability of having allergies is 10% and the probability of having sinus trouble if one has allergies but not the flu is 20%.

Now let's turn to a more complicated example. Suppose an alarm is installed in your home, and the alarm (A) can be set off by an earthquake (E) or a burglar (B). If the alarm goes off, it might cause your neighbor to call (C). Finally, if there is an earthquake, it might be reported on the radio (R). All in all, we have five binary random variables: A, E, B, C, and R. We will represent this domain with the following Bayes net:



The burglary variable can be either true or false, but it isn't "caused" by any other variable in the network. The same goes for the earthquake. The alarm depends on whether or not there is an earthquake, and whether or not there is a burglary. The neighbor's call depends on the alarm, and the radio depends on the earthquake. The parents of a node, intuitively, are the only things which directly influence that node.

We'll annotate this graph with the following probability distributions:

- $\mathrm{P}(B)$ — the prior probability of a burglary.
- $\mathrm{P}(E)$ — the prior probability of an earthquake.
- $\mathrm{P}(A \mid B, E)$ — the probability of the alarm going off under any of the relevant circumstances: (b^1, e^1), (b^1, e^0), (b^0, e^1), (b^0, e^0).
- $\mathrm{P}(C \mid A)$ — the probability of the neighbor's call for each value of A.
- $\mathrm{P}(R \mid E)$ — the probability of the radio reporting an earthquake given each value of E.

In particular, here are our CPTs:

$\mathrm{P}(B)$:

b^0	b^1
0.99	0.01

$\mathrm{P}(E)$:

e^0	e^1
0.995	0.005

$\mathrm{P}(A \mid B, E)$:

		a^0	a^1
b^0	e^0	0.999	0.001
b^0	e^1	0.7	0.3
b^1	e^0	0.2	0.8
b^1	e^1	0.05	0.95

$\mathrm{P}(R \mid E)$:

	r^0	r^1
e^0	0.99999	0.00001
e^1	0.65	0.35

$\mathrm{P}(C \mid A)$:

	c^0	c^1
a^0	0.95	0.05
a^1	0.3	0.7

Now, suppose we are given the state $(b^1, e^0, a^1, c^1, e^0)$. What is the probability of this exact state? We define it as follows:

$$\begin{aligned}
\mathrm{P}(b^1, e^0, a^1, c^1, e^0) &= \mathrm{P}(b^1)\mathrm{P}(e^0)\mathrm{P}(a^1 \mid b^1, e^0)\mathrm{P}(c^1 \mid a^1)\mathrm{P}(r^0 \mid e^0) \\
&= 0.1 \times 0.995 \times 0.8 \times 0.7 \times 0.99999 \\
&= 0.05572.
\end{aligned}$$

We just multiplied together the corresponding entries of all of our conditional probability tables.

We can state this definition for general Bayesian networks as follows. Each node X_i associated with a CPT $\mathrm{P}(X_i \mid \mathrm{Parents}(X_i))$ which specifies a distribution over X_i for each combination of values for X_i's parents. A

Bayesian network represents a joint probability distribution over its variables $X_1, \ldots, X_n$ via the **chain rule** for Bayes nets:

$$\mathrm{P}(x_1, \ldots, x_n) \;=\; \prod_{i=1}^{n} \mathrm{P}(x_i \mid \mathrm{Parents}(X_i)). \tag{1}$$

Bayesian networks give us a compact way of specifying the joint distribution over a set of variables. In our alarm network, we have five variables, each of which can take on one of two values. Therefore, explicitly representing the joint distribution over these variables in a table would require specifying 32 entries. The only constraint is that they all sum to 1, and so we need to specify 31 parameters. On the other hand, by representing the domain as a Bayes net, we only need to specify 10 parameters: 1 for $\mathrm{P}(B)$, 1 for $\mathrm{P}(E)$, 4 for $\mathrm{P}(A \mid B, E)$, 2 for $\mathrm{P}(C \mid A)$, and 2 for $\mathrm{P}(R \mid E)$.

4 Reasoning patterns

We now discuss some particular kinds of queries we can pose using Bayes nets. Suppose we have two subsets of variables: $\mathbf{Q}$, the **query**, and $\mathbf{E}$, the **evidence**. We are interested in computing the probability of the query given the evidence. (For instance, we might be interested in computing the probability that there was a burglary (b^1) given that our neighbor did not call (c^0). Then $\mathbf{Q} = \{B\}$ and $\mathbf{E} = \{C\}$.) We have defined the joint probability of all of the variables in terms of the CPT entries for each of the nodes in the Bayes net. Therefore, in principle, we can answer this query by explicitly writing out the full joint distribution in a table, and then marginalizing out over all of the irrelevant variables. (We'll discuss better methods later in these notes.) In the example above, we can compute:

$$\begin{aligned} \mathrm{P}(b^1 \mid c^0) &= \frac{\mathrm{P}(b^1, c^0)}{\mathrm{P}(c^0)} \\ &= \frac{\sum_{a,e,r} \mathrm{P}(b^1, c^0, a, e, r)}{\sum_{a,e,r,b} \mathrm{P}(c^0, a, e, r, b)} \end{aligned}$$

We'll now present some informal terms for various kinds of reasoning in Bayes nets:

- **Causal reasoning**. What is the chance that we get a phone call from our neighbor given that there was a burglary? In this case, the query is "downstream" of the evidence.

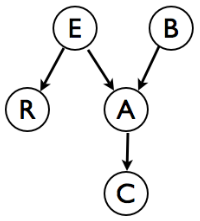


Figure 1: The alarm network. There are five nodes: E (earthquake), B (burglary), A (alarm), C (neighbor's call), and R (radio report of an earthquake).

- **Diagnostic** or **evidential reasoning**. Given that the alarm went off, what is the chance that there was a burglary? Here, we are trying to infer the probability of upstream events conditioned on downstream events.

- **Intercausal reasoning** or **explaining away**. Suppose your neighbor calls and informs you that your alarm went off. You are worried that there was a burglary. Then, you hear on the radio that there was an earthquake, and you're relieved because you figure the earthquake probably set off the alarm. We say the earthquake explained away the alarm. This is a very sophisticated form of reasoning, but we will see later that it fits nicely into the Bayes net framework.

5 Bayesian network semantics

We have seen that Bayes nets give a compact way of representing the joint distribution over a set of random variables. Specifying the full distribution over the five binary variables in our alarm network (shown again in Figure 1) by listing all of the individual probabilities in a table required specifying 31 parameters. Specifying the CPT entries for a Bayes net required only 10 parameters. Clearly, since it has fewer parameters to set, the latter approach can only specify a subset of the possible joint distributions over those five variables. We will now formalize precisely which subset of the joint distributions are consistent with a given Bayes net structure.

We will see that a Bayes net is basically encoding **conditional independence** assumptions about the variables in the network. Recall that we have committed to the following joint distribution over variables in the alarm network:

$$\mathrm{P}(B, E, A, C, R) = \mathrm{P}(B)\mathrm{P}(E)\mathrm{P}(A \mid B, E)\mathrm{P}(C \mid A)\mathrm{P}(R \mid E),$$

or for the more general case, we have the **chain rule for Bayes nets**:

$$\mathrm{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathrm{P}(X_i \mid \mathrm{Parents}(X_i)). \tag{2}$$

Any distribution consistent with the Bayes net must factorize in this way. In principle, we can find all of the conditional independencies in the network simply by performing algebraic manipulations on this expression. For instance, as an exercise, try to prove that B and E are independent in the alarm network. It is clearly tedious to try to uncover all of the conditional independencies through brute force algebraic manipulation. We will now develop higher-level sufficient and necessary conditions for conditional independence which allow us to read off conditional independencies directly from the graph structure.

First, let us recall the definition of conditional independence: Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ be (not necessarily disjoint) sets of random variables. (Recall that we use plaintext to denote random variables and their values, and boldface to denote sets of random variables and their possible joint assignments.) We define:

$\mathbf{X}$ is **conditionally independent** of $\mathbf{Y}$ given $\mathbf{Z}$ if $\mathrm{P}(\mathbf{x} \mid \mathbf{y}, \mathbf{z}) = \mathrm{P}(\mathbf{x} \mid \mathbf{z})$
for all assignments $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ to $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$.

Equivalently, $\mathbf{X}$ and $\mathbf{Y}$ are conditionally independent given $\mathbf{Z}$ if $\mathrm{P}(\mathbf{x}, \mathbf{y} \mid \mathbf{z}) = \mathrm{P}(\mathbf{x} \mid \mathbf{z})\mathrm{P}(\mathbf{y} \mid \mathbf{z})$. As a shorthand, we often write this as $\mathrm{P}(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z}) = \mathrm{P}(\mathbf{X} \mid \mathbf{Z})\mathrm{P}(\mathbf{Y} \mid \mathbf{Z})$. We will also write $\mathrm{I}(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z})$ to denote that $\mathbf{X}$ is conditionally independent of $\mathbf{Y}$ given $\mathbf{Z}$.

Before we proceed to precisely define necessary and sufficient conditions for conditional independence, let us consider some simple cases. Figure 2 shows some examples where influence does or does not flow from one node X to another node Y. Make sure you understand why influence does or does not flow in each of these cases.

6 d-separation

Now we are going to formalize these intuitions. Suppose we have a three-variable path $X - Z - Y$ in our network. We say this path is **active** (influence "flows" from X to Z through Y) if one of the following holds:

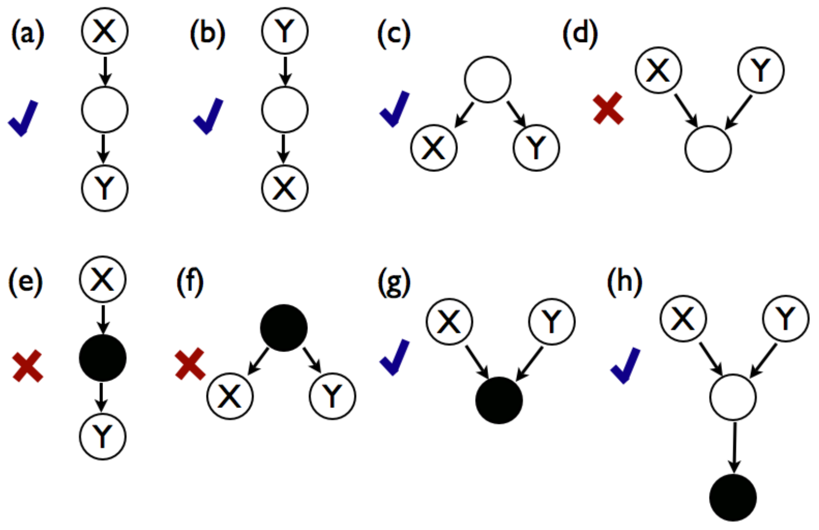


Figure 2: Some examples of conditional independence. In each of these networks, white nodes are unobserved and black nodes are observed. When influence flows from X to Y (i.e. X is not conditionally independent of Y given the observed nodes), the graph is marked with a check. Otherwise, it is marked with an X. (a) Knowing there was a burglary makes it more likely that our neighbor calls. (b) Knowing our neighbor called makes it more likely there was a burglary. (c) Knowing the alarm went off makes it more likely that an earthquake will be reported on the radio. (d) Knowing there was a burglary doesn't tell us anything about whether there was an earthquake. (e) Knowing there was an earthquake doesn't tell us anything about whether our neighbor calls, if we already know the alarm went off. (f) Knowing an earthquake was announced on the radio doesn't tell us anything about whether the alarm went off, if we already know there was an earthquake. (g) If we know the alarm went off, then knowing there was an earthquake "explains away" the alarm, making it less likely there was a burglary. (h) If we know our neighbor called, then knowing there was an earthquake explains away the call, making it less likely there was a burglary.

1. Z is not observed, and the graph structure is one of the following:
$$X \to Z \to Y, \quad X \leftarrow Z \to Y, \quad \text{or} \quad X \leftarrow Z \leftarrow Y.$$
2. The graph structure is a **V-structure** (i.e. $X \to Z \leftarrow Y$) and Z, or some descendant of Z, is observed.

Now let's generalize this definition to longer paths. A path $X_1, X_2, \dots, X_k$ is **active** given a set of observed nodes $\mathbf{Z}$ if:

1. For any V-structure $X_i \to X_{i+1} \leftarrow X_{i+2}$, either the vertex X_{i+1} is observed, or some descendant of X_{i+1} is observed.
2. No other node on the path is observed.

Two nodes X and Y are **d-separated** in a graph G given $\mathbf{Z}$ if there is no active path between them in G. Note that we are defining d-separation in terms of the graph structure, rather than the underlying distribution. To connect the graph structure and the distribution, we will need the following theorem, which we state without proof:

Theorem 6.1: *Let* P *be a distribution that factors according to the Bayes net structure given by the graph G. Suppose two nodes X and Y are d-separated in G given a set of nodes* $\mathbf{Z}$*. Then X and Y are conditionally independent given* $\mathbf{Z}$*.*

Because of this theorem, we say d-separation is a **sound** mechanism for inferring conditional independence. I.e., if two nodes X and Y are d-separated, then they are necessarily conditionally independent given $\mathbf{Z}$. It turns out that d-separation is also an "almost complete" mechanism for inferring conditional independence, in a sense that we now explain. If there is an active path between two nodes X and Y given $\mathbf{Z}$, does this mean X and Y are necessarily conditionally dependent given $\mathbf{Z}$? No, because we could assign probabilities in the CPTs in such a way that X and Y *happen* to be conditionally independent given $\mathbf{Z}$. However, the following is true:

Theorem 6.2: *If two nodes X and Y are* not *d-separated in G given* $\mathbf{Z}$*, there will be some choice of CPT entries such that X and Y are not independent given* $\mathbf{Z}$*.*

Hence, in a limited sense, d-separation is also **complete**. With d-separation, we can prove many statements about independencies directly from the graph structure, rather than by directly manipulating the terms in the definition (2).

7 Bayes ball

We have just specified necessary and sufficient conditions for two random variables X and Y being conditionally independent given some set of observed variables $\mathbf{Z}$. These conditions are useful in mathematical proofs about Bayes nets. However, we are sometimes interested in computing, for a particular graph, which variables are conditionally independent of which other variables.

We now present the **Bayes ball** algorithm, an efficient method for identifying all of the variables in the network which influence X.[1] We discuss the algorithm as if we were carrying it out by hand, but it is possible to formalize Bayes ball as an efficient dynamic programming algorithm. We imagine we have a ball which begins at node X, and which can travel anywhere that influence flows. If there is any way for the ball to travel from X to Y, then X and Y are conditionally dependent given $\mathbf{Z}$.[2] Otherwise, they are conditionally independent given $\mathbf{Z}$.

Now let's specify how the ball is allowed to bounce. We use the rules shown in Figure 3. As before, we use white circles to denote unobserved nodes and black circles to denote observed nodes. In cases where the ball may pass through, we draw red arrows going both directions; in cases where it cannot pass through, we draw arrows which turn around.

These rules have clear similarities with d-similarity. For instance, the ball can pass through a path $X \to Z \to Y$ if and only if Z is unobserved. However, there is one subtle difference. Notice that the Bayes Ball rules for V-structures say nothing about whether a descendant of the vertex is observed; they only require knowing whether the particular nodes on the path are observed. Consider the problem of showing that R is conditionally dependent on B given C, as demonstrated in Figure 4. Using the properties of d-separation, we find the active path marked in Figure 4(a). However, the Bayes ball path will be the one given in Figure 4(b). Note that the rule which allows the ball to "bounce" back up when it hits C is the one for $X \to Z \leftarrow Y$, where Z is observed, and X *and* Y *are identical.* (In Bayes ball, unlike d-separation, we allow repeated nodes in the paths.) It is this locality of the rules which allows Bayes ball to efficiently discover the conditional dependencies in the graph.

[1]This algorithm was not covered in the class lectures, and the material in this section is optional, and will not directly appear in CS221 homeworks/midterm/etc.

[2]More formally, there is some assignment to the CPT entries such that X and Y are conditionally dependent given $\mathbf{Z}$.

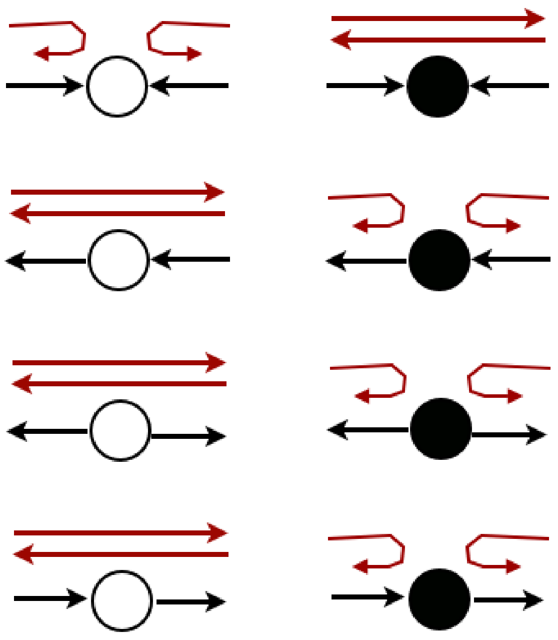


Figure 3: Rules for Bayes ball. We use white circles to denote unobserved nodes and black circles to denote observed nodes. In cases where the ball may pass through, we draw red arrows going both directions; in cases where it cannot pass through, we draw arrows which turn around.

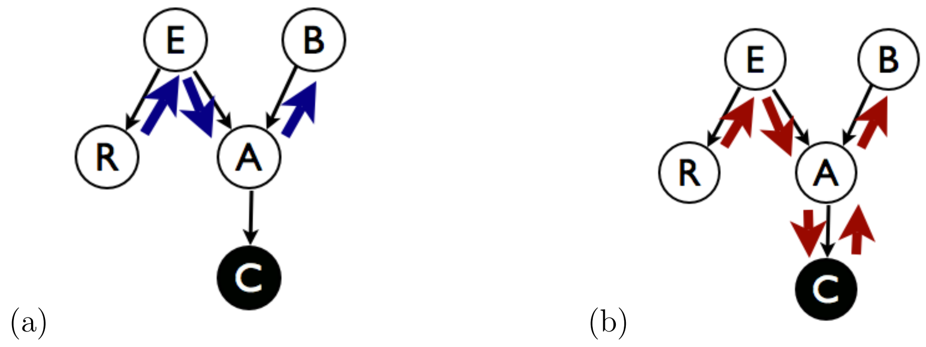


Figure 4: An example of the difference between d-separation and Bayes ball. (a) The active path from R to B given C. (b) The path the Bayes ball takes from R to B.

8 Bayes net inference

Given a Bayes net structure of a domain, we often want to perform **inference** on the Bayes net. Specifically, we might want to determine the conditional probability $P(q \mid \mathbf{e})$ that some **query** variable Q takes on a value q given that another set of variables $\mathbf{E}$, the **evidence** variables, has a joint assignment $\mathbf{e}$. We saw previously that we can, in principle, compute this conditional probability by explicitly writing out the full joint distribution over all of the variables in a big table, computing the entries using the chain rule for Bayes nets

$$\mathrm{P}(x_1, \ldots, x_n) = \prod_{i=1}^{n} \mathrm{P}(x_i \mid \mathrm{Parents}(X_i)),$$

and then applying the definition of conditional probability,

$$\mathrm{P}(q \mid \mathbf{e}) = \frac{\mathrm{P}(q, \mathbf{e})}{\mathrm{P}(\mathbf{e})}.$$

However, the size of such a table would be exponential in the number of variables in the network, and therefore this algorithm is prohibitively expensive for all but the smallest networks. We will show how to use the structure of Bayes nets to perform inference more efficiently.

Suppose we have a two-variable Bayes net $A \rightarrow B$, where A and B are both binary random variables, and we want to compute the marginal probability of B, $\mathrm{P}(B)$. We can apply our formula for marginal probability:

$$\mathrm{P}(b^1) = \mathrm{P}(a^0)\mathrm{P}(b^1 \mid a^0) + \mathrm{P}(a^1)\mathrm{P}(b^1 \mid a^1). \tag{3}$$

Each of the terms is just one of the entries in the CPTs for the Bayes net. We can compute $P(b^0)$ similarly. Recall that we introduced a shorthand notation where uppercase variables in an equation signify that the equation must hold true for any value of the random variable. Using this notation, we can rewrite (3) as:

$$\mathrm{P}(B) = \mathrm{P}(a^0)\mathrm{P}(B \mid a^0) + \mathrm{P}(a^1)\mathrm{P}(B \mid a^1).$$

More generally, when A is not binary, we have:

$$\mathrm{P}(B) \;\; = \;\; \sum_{a} \mathrm{P}(a)\mathrm{P}(B \mid a^0). \tag{4}$$

If A and B can each take on k values, this sum can be computed in $O(k^2)$ time (i.e. $O(k)$ for each value in the domain of B).

Now let's consider a longer chain: $A \to B \to C$. We want to compute $\mathrm{P}(C)$. If we knew the marginal distribution $\mathrm{P}(B)$, we could compute $\mathrm{P}(C)$ in the same way as above:

$$\mathrm{P}(C) = \sum_b \mathrm{P}(b)\mathrm{P}(C \mid b).$$

But how do we get $\mathrm{P}(B)$? We find it in the same way:

$$\mathrm{P}(B) = \sum_a \mathrm{P}(a)\mathrm{P}(B \mid a).$$

Therefore, finding $P(C)$ only requires applying (4) twice. More generally, given a chain $X_1, X_2, \ldots, X_n$, we apply (4) $n-1$ times. Therefore, the total running time will be $n-1$ times the running time for computing (4), or $O(nk^2)$. By contrast, if we had tried to compute this sum the naive way (by explicitly writing out the full joint distribution), we would have had to produce a table with $O(k^n)$ entries. We have gone from exponential time to linear time in the number of variables.

Let's formalize this process. Suppose we have the chain Bayes net $A \to B \to C \to D$, and we want to compute $\mathrm{P}(D)$. By definition,

$$\begin{aligned}
\mathrm{P}(d) &= \sum_{a,b,c} \mathrm{P}(a)\mathrm{P}(b \mid a)\mathrm{P}(c \mid b)\mathrm{P}(d \mid c) \\
&= \sum_c \sum_b \sum_a \mathrm{P}(a)\mathrm{P}(b \mid a)\mathrm{P}(c \mid b)\mathrm{P}(d \mid c) \\
&= \sum_c \mathrm{P}(d \mid c) \sum_b \mathrm{P}(c \mid b) \sum_a \mathrm{P}(a)\mathrm{P}(b \mid a). \qquad (5)
\end{aligned}$$

Consider only the term $\sum_a \mathrm{P}(a)\mathrm{P}(b \mid a)$ in (5). The value of this term depends on the value of B. Therefore, we can rewrite the term as a **factor** $f_1(B)$, or a table of k numbers, one for each value of B. Specifically,

$$f_1(b) = \sum_a \mathrm{P}(a)\mathrm{P}(b \mid a).$$

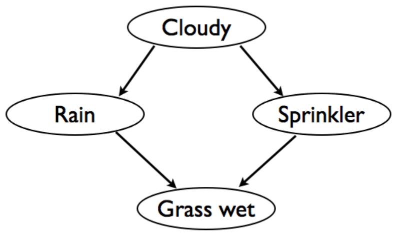
In this particular case, $f_1(b)$ turns out to be the marginal probability of $B = b$, but be aware that this won't hold true in general. *The factors we discuss here will not always correspond to marginal or conditional probabilities,* and we won't discuss their intuitive meaning any further. Once we have the factor $f_1(B)$, we can plug it into (5):

$$\mathrm{P}(d) \;=\; \sum_c \mathrm{P}(d \mid c) \sum_b \mathrm{P}(c \mid b) f_1(b). \qquad (6)$$

Just as we did before, we can compute the sum $\sum_b P(c \mid b) f_1(b)$ to get another factor $f_2(C)$, containing one number for each value of c. Finally, we plug $f_2(C)$ into (6) to get

$$P(d) = \sum_c P(d \mid c) f_2(c).$$

Let us consider one final example before we define variable elimination. Suppose we are trying to predict whether our neighbor's grass will be wet. Assume we have the following Bayes net structure:



If it is cloudy, it is more likely to rain. If our neighbors see that it is not cloudy, they are more likely to decide to turn on the sprinkler. Finally, either rain or the sprinkler being on can explain the wet grass.

Like before, we can express $P(W)$ as follows:

$$\begin{aligned} P(w) &= \sum_{r,s,c} P(w, r, s, c) \\ &= \sum_{r,s,c} P(w \mid r, s) P(r \mid c) P(s \mid c) P(c) \\ &= \sum_{r,s} P(w \mid r, s) \sum_c P(r \mid c) P(s \mid c) P(c). \end{aligned}$$

Let's first sum out the terms over C to get a factor $f_1(r, s)$, a table containing one value for each pair (r, s). Then we compute:

$$P(w) = \sum_{r,s} P(w \mid r, s) f_1(r, s).$$

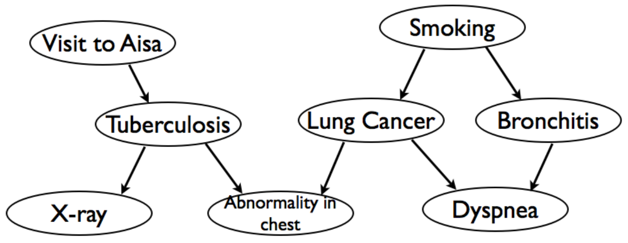
To summarize: the joint distribution of all of the variables in a Bayes net is defined by the chain rule for that Bayes net. By judiciously choosing subexpressions to compute first, we can avoid the exponential blowup that would occur if we tried to write out the entire joint distribution in a table.

8.1 Variable elimination

Now, let's generalize what we did in these examples into the **variable elimination** algorithm. Suppose we are trying to compute the marginal probability $P(q)$ of a query variable q.[3]

> Let $X_1, X_2, \ldots, X_m$ be an ordering of the non-query variables (i.e. the variables other than q).
>
> Consider the chain rule over the network structure:
>
> $$\sum_{x_1} \sum_{x_2} \cdots \sum_{x_m} \prod_{j=1}^{n} P(x_j \mid \text{Parents}(X_j)).$$
>
> For $i = 1, \ldots, m$:
>
> > Leave in the summation for X_i only the factors which mention X_i.
> >
> > Multiply out all of these factors, getting a factor f that contains a number for each of the possible joint assignments to the variables mentioned, including X_i.
> >
> > Sum out the factor f over X_i, getting a factor f' which contains a number for each of the possible joint assignments, not including X_i.
> >
> > Replace the sum $\sum_{x_i} \prod \cdots$ with the factor f'.

Now let's consider a more complex example of variable elimination. Consider the following network:



[3]We stated earlier that we often want to compute $P(q \mid \mathbf{E})$, the probability of the query given the evidence. Here, we suppose we do this by computing $P(q, \mathbf{E})$ and $P(\mathbf{E})$, and then taking the ratio. However, with small modifications, we can use variable elimination to compute $P(q \mid E)$ directly. This is often significantly faster than computing both $P(\mathbf{E})$ and $P(q, \mathbf{E})$.

Suppose we want to compute the probability of dyspnea. The joint distribution is defined as:

$$\mathrm{P}(d^1) = \sum_{a,b,\ell,t,x,s,v} \mathrm{P}(v)\mathrm{P}(t \mid v)\mathrm{P}(s)\mathrm{P}(\ell \mid s)\mathrm{P}(b \mid s)\mathrm{P}(a \mid \ell, t)\mathrm{P}(x \mid t)\mathrm{P}(d^1 \mid l, b).$$

First, we eliminate V to get the factor

$$f_v(t) = \sum_V \mathrm{P}(t \mid v)\mathrm{P}(v).$$

For instance, if T can take on the values *no*, *mild*, or *severe*, f_V will be a factor with three numbers. In the next step, we eliminate S to get:

$$f_S(b, c) = \sum_s \mathrm{P}(s)\mathrm{P}(\ell \mid s)\mathrm{P}(b \mid s).$$

If bronchitis has 3 values (*no*, *mild*, and *severe*) and lung cancer has 4, this gives a table of 12 numbers. We continue this process until we have eliminated all of the variables in the network (besides D).

What is the running time of variable elimination? The inner loop requires constructing a factor over some set of variables $\mathbf{A}$, and then marginalizing this factor with respect to one of the variables. The time required to do this will be roughly linear *in the size of the factor*. For instance, if all of the variables are binary and we produce a factor over four variables, that factor will have 16 entries. Since the inner loop is executed once for each non-query variable, the running time will be $O(mk^D)$, where m is the number of non-query variables, k is the size of the largest domain of any variable, and D is the largest number of variables mentioned in any factor produced by variable elimination. This is exponential in k, but Bayes net inference is NP-hard, and so (assuming $P \neq NP$) the worst-case complexity will be exponential in the size of the network, no matter what algorithm we use.[4]

Given that variable elimination's running time is exponential in the size of the largest factor, how large will the factors be in practice? As a rule of thumb, Bayes nets which are tightly connected (in that there are many active paths through which one variable influences another) will tend to produce large intermediate factors. If the network is sparsely connected, the factors will tend to remain small. Also, the size of the intermediate factors is heavily dependent on the particular variable ordering we choose. Typically, we choose the ordering by hand. Finding the best ordering is NP-hard in general, but several heuristics work well in practice. You will see examples of this in section.

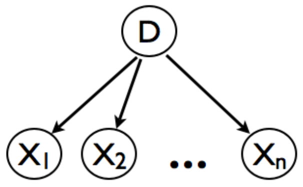
[4]More specifically, the worst-case complexity is exponential in the number of CPT entries needed to specify the joint distribution.

CS221 Lecture notes #12

Bayesian networks, continued

1 Applications

Bayes nets have been applied to a wide variety of problems, and here we outline just a few. An early example was PATHfinder, a system which diagnosed pathologies in lymph nodes. The earliest versions of this system were based on a system of formal rules, but later versions used Bayes nets. The first Bayes net structure that was incorporated into PATHfinder is called **Naive Bayes**. A Naive Bayes network for medical diagnosis has a single node D which represents whether or not the patient has a particular disease, and all of the other variables $X_1, X_2, \ldots, X_n$ are direct children of the disease node. These variables represent symptoms, and we suppose that all symptoms are independent given the presence or absence of the disease. Here is such a network:



Applying our general definition of Bayes nets, the joint distribution over all of the variables is given by:

$$\mathrm{P}(D, X_1, \ldots, X_n) = \mathrm{P}(D) \prod_{i=1}^{n} \mathrm{P}(X_i \mid D).$$

This structure makes it easy to compute the conditional probability of a disease given the presence or absence of each of the symptoms:

$$\begin{aligned}
\mathrm{P}(d^1 \mid x_1, \ldots, x_n) &= \frac{\mathrm{P}(d^1, x_1, \ldots, x_n)}{\mathrm{P}(x_1, \ldots, x_n)} \\
&= \frac{\mathrm{P}(d^1, x_1, \ldots, x_n)}{\mathrm{P}(d^1, x_1, \ldots, x_n) + \mathrm{P}(d^0, x_1, \ldots, x_n)} \\
&= \frac{\mathrm{P}(d^1)\mathrm{P}(x_1 \mid d^1)\cdots\mathrm{P}(x_n \mid d^1)}{\mathrm{P}(d^1)\mathrm{P}(x_1 \mid d^1)\cdots\mathrm{P}(x_n \mid d^1) + \mathrm{P}(d^0)\mathrm{P}(x_1 \mid d^0)\cdots\mathrm{P}(x_n \mid d^0)}
\end{aligned}$$

The conditional probability of a disease given its symptoms can, therefore, be computed in linear time.

The next version of the PATHfinder network eliminated 10% of all of the misdiagnoses of this network just by eliminating all of the CPT entries which were assigned the value 0. No amount of evidence can make an event seem possible if it had a prior probability of zero. More generally, unless an event is absolutely impossible, it is usually a bad idea to assign it a CPT entry of 0 in a Bayes net.

Later, PATHfinder was expanded into a full Bayes net, which could take into account not only symptoms, but other factors such as family history or behavior, which could affect the prior probability of having a disease. The overall results with this full Bayes net were equivalent to saving 1 life in 1000.

PATHfinder was shown to outperform human pathologists in many situations because:

- Bayes nets incorporate the prior probabilities of various diseases in a principled way. Often, people have trouble precisely weighting prior probabilities against the evidence. For example, psychology studies have shown that human physicians tend to weight the prior probability less than they should. Instead, they tend to focus on the probability of the symptoms given the disease, thereby assigning high likelihoods to very rare diseases.

- Bayes nets are better at incorporating all of the different pieces of evidence available. Humans have a hard time keeping more than 7-9 pieces of evidence in their heads at a time, while Bayes nets can easily consider dozens of pieces of evidence.

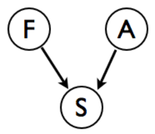
With Bayes nets, one can also determine which further piece of evidence would be most useful to observe, possibly taking into account that it may cost different amounts to observe different variables. In medical diagnosis, this helps avoid unnecessary medical tests, thereby saving time and money, and possible even sparing the patient from painful procedures.

As another example, Microsoft uses a Bayes net to diagnose printer errors. Using a Bayes net rather than a set of hard-coded rules allows the system to make good predictions even if the user chooses not to enter particular information, such as the results of printing out a test page.

2 Parameter learning

We have discussed the semantics of Bayes nets and how to perform inference. We now discuss how to come up with the Bayes net in the first place. There are actually two problems: **structure learning** (determining the BN graph structure) and **parameter learning** (assigning values to the CPT entries). Typically, we specify a Bayes net structure by hand rather than use a learning algorithm. There are various algorithms for learning Bayes net structures from data, but it is far more common to hand-specify the graph structure, and so we won't discuss these algorithms here. Instead, we will focus on the more important problem of parameter learning.

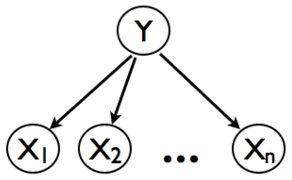
Let's return to our earlier flu network example. Recall that we had three random variables: flu (F), allergy (A), and sinus trouble (S). The network structure was as follows:



Suppose we have a database of patients, and we know the values of F, S, and A for each of them. Then, we would estimate $\mathrm{P}(f^1)$ as the fraction of people in this database who had the flu. To estimate $\mathrm{P}(s^1 \mid f^1, a^1)$, we would use the fraction of people with the flu and allergies who had a sinus infection.[1]

Finally, it's worth noting that the Naive Bayes network mentioned above is often used as a supervised learning algorithm. More specifically, assume we have n discrete-valued random variables $X_1, X_2, \ldots, X_n$, and we are trying to predict the value of a discrete-valued target variable Y. Suppose we have a training set $S_{\text{train}} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$. We apply the Naive Bayes network structure:

[1]As an exercise, try to justify this method using maximum likelihood.



We estimate all of the parameters for this network just as we did above. Then, given a new example $X = (X_1, X_2, \ldots, X_n)$, we predict

$$\begin{aligned}\arg\max_y \mathrm{P}(y \mid x_1, \ldots, x_n) &= \arg\max_y \frac{\mathrm{P}(y, x_1, \ldots, x_n)}{\mathrm{P}(x_1, \ldots, x_n)} \\ &= \arg\max_y \mathrm{P}(y, x_1, \ldots, x_n) \\ &= \arg\max_y \mathrm{P}(y) \prod_{i=1}^{n} \mathrm{P}(x_i \mid y). \qquad (1)\end{aligned}$$

Finally, an implementation note. If we were to compute (1) directly, we might have problems with numerical underflow, especially if any of the probabilities are very small. Instead, we take the logarithm of each of the terms, and choose as our prediction:

$$\arg\max_y \log \mathrm{P}(y) + \sum_{i=1}^{n} \log \mathrm{P}(x_i \mid y).$$

This can be safely computed without numerical underflow.

Naive Bayes often does not perform as well as logistic regression or a well-designed decision tree, but it is such a simple algorithm that it is worth using in many cases.

CS221 Lecture notes #13

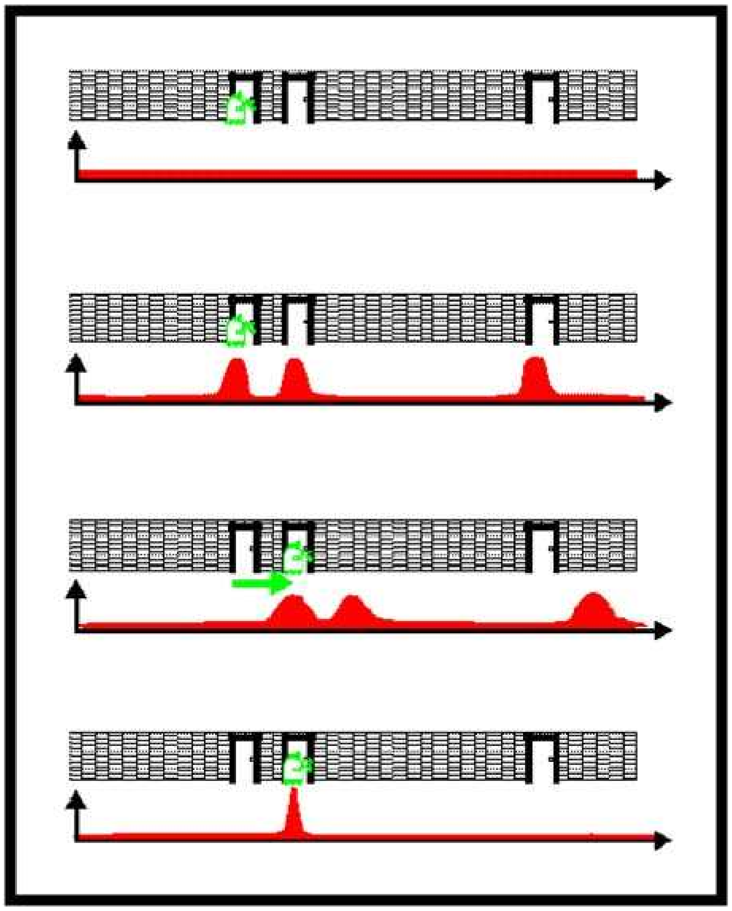
Hidden Markov Models

There is a number of algorithms that predate the growth Bayes nets, but later became understood as a specific kind of Bayes net structure. Naive Bayes is one example. Another example is the **Hidden Markov Model (HMM)**, a probabilistic model for representing variables that evolve over time.

1 The robot localization problem

HMMs are widely used for a range of very different applications. But to explain them, we'll focus on the specif problem of robot localization. In this problem, the robot has a map of its environment and a collection of sensors. The robot's belief about where it is can be represented as a probability distribution over locations on the map. Initially the robot does not know where it is. This can be represented by a uniform probability distribution. As the robot begins to move around and collect observations, the distribution will become peaked around locations that are consistent with the sensor readings that the robot observes.

To make this more concrete, let's consider a specific example, illustrated in this figure:



(Figure courtesy of Thrun, Burgard and Fox.)

In this example, the robot, shown in green, can only move in one dimension, along a hallway. The robot's only sensor is a door detector, which observes whether the robot is in front of a door or not. The robot does not know where it is initially, but it does know that it is in the hallway and it has a map of the hallway. Initially the robot's beliefs about its location, shown in red, is uniform distribution over all position in the entire hallway. It then observes that it is in front of a door, causing the distribution representing its belief to become peaked in front of the location of each of the doors on the map. The robot then starts to move to the right. Because the robot knows that it is moving, the peaks in its belief distribution move rightward as well. The peaks flatten somewhat to represent uncertainty in exactly how far the robot has moved. A little later, the door detector tells the robot again that it is in front of a door. At this instant, only one of the three peaks is in front of a door, so the other two become very unlikely. After this observation, the robot's belief distribution is sharply peaked at its true location.

For a real robot, the situation is slightly more complicated. The robot will have a two dimensional map of the building. Additionally, the robot will have a collection of range-finding sensors that measure the distance from the robot to obstacles in several directions simultaneously. These sensors may be such sensors as sonar or lasers range scanners.

2 Hidden Markov Models for robot localization

To formally model the problem described above, we will define a probability distribution over the following random variables:

- S_t = state of the robot at time t. If the robot's state is just a two-dimensional location (x, y), we can discretize the state space using a two-dimensional grid. (In the more general case, we may also wish to model the robot's orientation θ.)

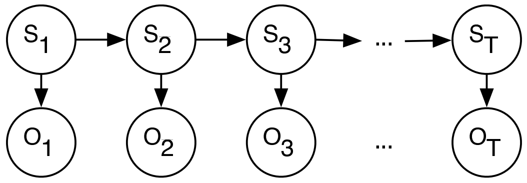
- O_t = observation at time t. In our setting, O_t will be vector-valued. It is the collection of all sensor measurements at time t.

We will also use the notation $O_{1:t} = \{O_1, \ldots, O_t\}$ to denote the set of all observations up to time t. As before, we will use upper-case to denote random variables, and lower-case to denote specific values that the random variables take on.

Given a specific sequence of observations $o_{1:t} = \{o_1, \ldots, o_t\}$ observed by the robot, our goal will be to compute

$$B(s_t) = P(s_t | o_{1:t})$$

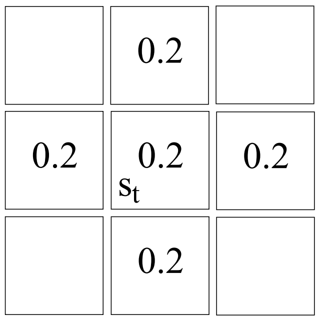
Our joint distribution over $S_{1:T}$ and $O_{1:T}$ will be defined using a Bayes net with the following graph structure:



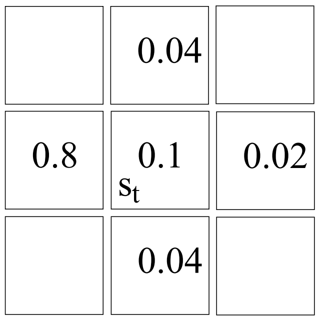
We also need to define the CPT for the Bayes net.

- $P(S_1)$ defines the distribution over where the robot is initially. Since we have no information about where the robot will begin, we set this CPT to the uniform distribution over all states.

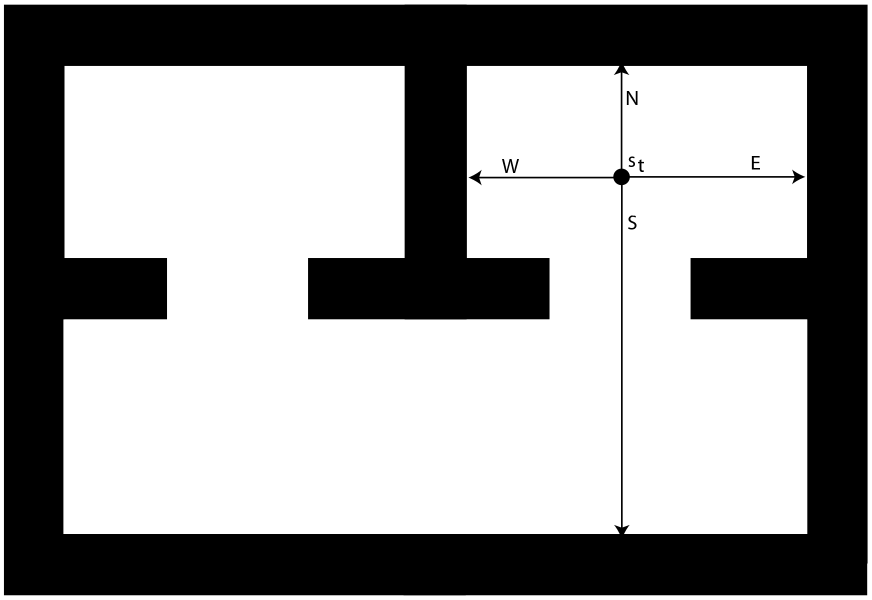
- $P(S_{t+1}|s_t)$ defines how the robot moves from one state to another, and is also referred to as the **state transition distribution**. Even though when the algorithm is running, we don't know precisely where the robot is, the Bayes net structure allows us to model the robot's movement with rules of the form "If the robot *were* at *precisely* state s in time t, then in time $t+1$ the robot's state would be distributed like this." As a simple example, this could encode a simple random walk: the robot has a .2 chance of moving one cell to the left, .2 chance of moving one cell to the right, and so on:



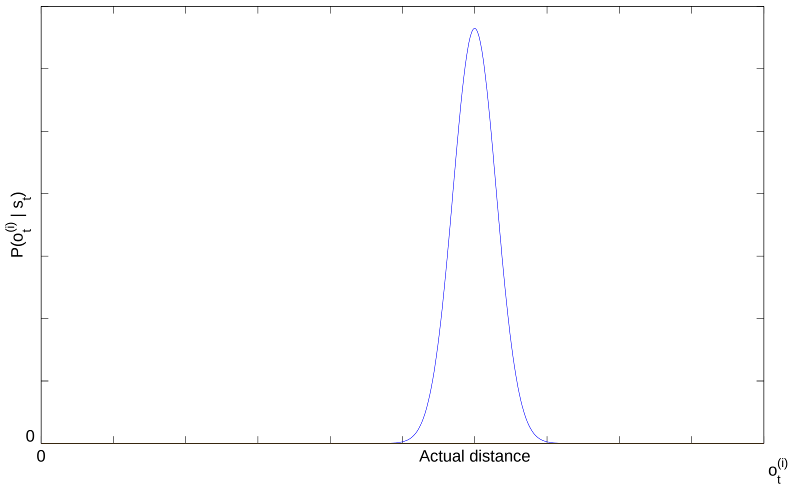
To make a more sophisticated model, we may also take into account the action a_t that the robot took on step step t, and model the robot's motion from state to state as $P(S_{t+1}|s_t, a_t)$. In this case the CPT could encode the idea that if the robot chose to move west at time t, then state s_{t+1} will be one grid cell west of s_t with 90% probability. For example, the distribution may look like this:



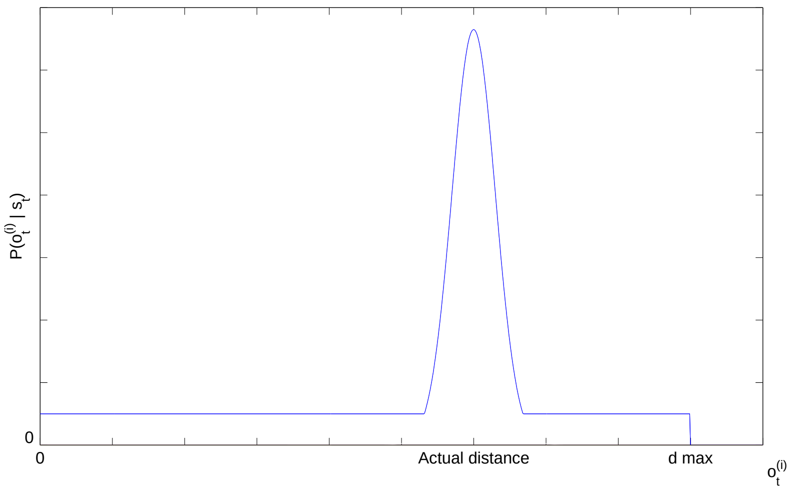
- $P(O_t|s_t)$ is the probability of making a certain set of observations when the robot is at a specific state, and is also called the **observation distribution**. In our setup, the observations are the distance measurements in different directions. Suppose that our robot has four distance sensors, pointed north, south, east and west. Then the measurements would be $O_t = \{O_t^{(1)}, O_t^{(2)}, O_t^{(3)}, O_t^{(4)}\}$, corresponding to the reported distances to the nearest wall in each of the four compass directions from the location s_t:



Of course the sensors have some noise, so we model the idea that each sensor returns *approximately* the true distance with a distribution that looks like this:



Additionally, because the sensor may sometimes fail entirely (or get a random measurement corresponding to someone walking in front of the robot, say), to get a better sensor model, we might also put a small positive probability everywhere in the interval $[0,\ d_{max}]$, where d_{max} is the maximum range of the sensor:



If we assume that errors in the sensors are independent, then

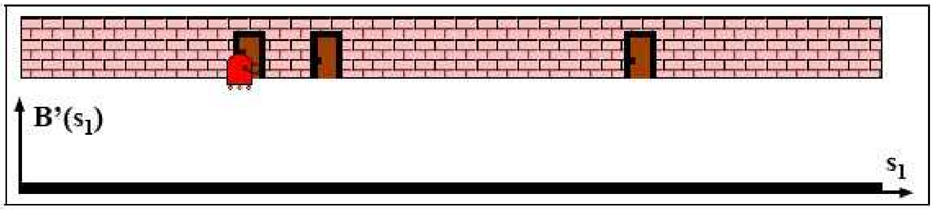
$$P(O_t | s_t) = \Pi_{i=1}^{4} P(O_t^{(i)} | s_t).$$

3 Inference algorithms

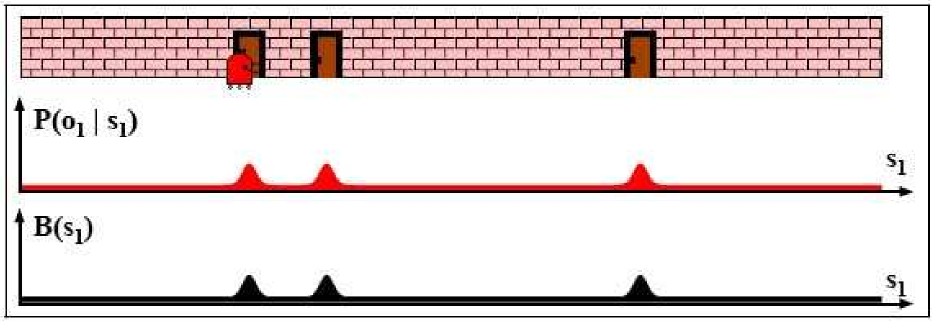
3.1 Overview of the filtering algorithm

In order to estimate the robot's location at time t, we need to compute $P(s_t | o_{1:t})$. We could do this using variable elimination, but it turns out that this would be very inefficient. The first value we would compute would be $P(s_1 | o_1)$. Next we would compute $P(s_2 | o_2, o_1)$, then $P(s_3 | o_3, o_2, o_1)$ and so on. Running a separate instance of variable elimination for each time step repeats many computations unnecessarily. Further, at time t of the robots life, to compute $P(s_t | o_{1:t})$ from scratch this way would require $O(t)$ time—so the robot would run slower and slower as t grows. Instead, here we will present a **filtering algorithm** that will allow us to use the beliefs from time t to compute the beliefs for time $t + 1$, without having to run inference along the entire chain of variables at each time step.

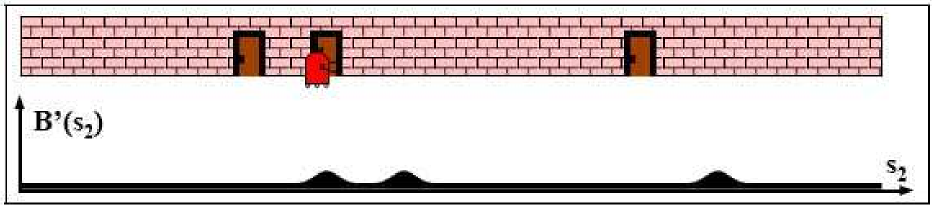
The following series of figures (also due to Thrun, Burgard, and Fox) help to illustrate the algorithm at a high level. First, we initialize the robot's beliefs about its location to a uniform distribution:



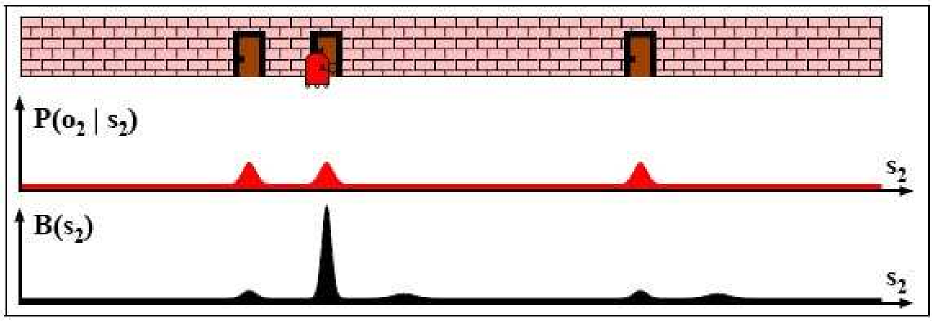
Next, the robot takes a sensor reading and performs the **observation update**. The observation update takes the information from the sensor reading and incorporates it into the robot's beliefs. Note that here, $P(o_1|s_1)$ is plotted as a function of s_1, not of o_1, since o_1 is an observed, known quantity.



The robot then moves down the hallway. The next step in the filtering algorithm is the **dynamics update** which takes account of this motion. After the dynamics update, the beliefs reflect the information that the robot has moved, but do not yet take any new sensor information into account:



Finally, we can perform another observation update to incorporate the new sensor information:



3.2 Details of the filtering algorithm

In the derivation below, we will assume throughout that we are using a discretized state representation. Thus, if $s = (x, y)$ is the position of the robot and we discretize the state space using a 10x10 grid, then s_t is a random variable that can take on any of 100 different values corresponding to the 100 different grid cells. (The case of $s = (x, y, \theta)$ discretized with a 3d grid is also handled similarly.)

The first value we will need to compute is

$$B(s_1) = P(s_1|o_1) = \frac{P(s_1, o_1)}{P(o_1)} = \frac{P(o_1|s_1)P(s_1)}{P(o_1)}$$

Note that both terms in the numerator can be obtained directly from the CPTs of the Bayes net. Further, because the denominator is constant with respect to s_1, and we know that

$$\sum_{s_1} B(s_1) = 1$$

We don't need to explicitly calculate $P(o_1)$. Instead, we can replace the denominator with a constant alpha:

$$B(s_1) = \alpha P(o_1|s_1)P(s_1)$$

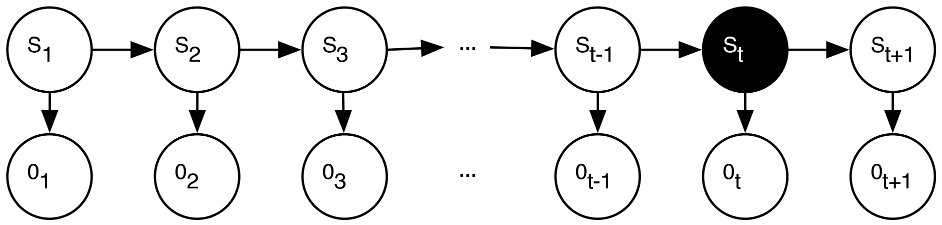
We can solve for α by making sure that $\sum_{s_1} B(s_1) = 1$.

Once we have our initial beliefs, $B(S_t)$, we can perform the dynamics update. In this step we compute $B'(s_{t+1})$, which represents our beliefs about where the robot will be after it makes its movement between time steps t and $t + 1$. $B'(s_{t+1})$ does not incorporate any information from o_{t+1} yet though.

It turns out that $B'(s_{t+1})$ can be computed using only our beliefs from time t and the transition model:

$$\begin{aligned} B'(s_{t+1}) &= P(s_{t+1}|o_{1:t}) \\ &= \sum_{s_t} P(s_{t+1}, s_t|o_{1:t}) \\ &= \sum_{s_t} P(s_{t+1}|s_t, o_{1:t})P(s_t|o_{1:t}) \\ &= \sum_{s_t} P(s_{t+1}|s_t)B(s_t) \end{aligned}$$

In the last step above, we used the the independence assumption $I(S_{t+1}, O_{1:t}|S_t)$. We can see that this assumption is implied by the d-separation properties of the Bayes net structure for the Hidden Markov model. When we observe s_t, there is no active path from $t-1$ or earlier to $t+1$:



After performing the dynamics update, we still need to use the new sensor information from time $t+1$ in the observation update. This update can be computed efficiently using only the observation model and $B'(s_t)$:

$$\begin{aligned} B(s_{t+1}) &= P(s_{t+1}|o_{1:t}, o_{t+1}) \\ &= \frac{P(o_{t+1}, s_{t+1}|o_{1:t})}{P(o_{t+1}|o_{1:t})} \\ &= \frac{P(o_{t+1}|s_{t+1}, o_{1:t})P(s_{t+1}|o_{1:t})}{P(o_{t+1}|o_{1:t})} \\ &= \frac{P(o_{t+1}|s_{t+1})B'(s_{t+1})}{P(o_{t+1}|o_{1:t})} \end{aligned}$$

In most implementations, we again do not explicitly compute the denominator. Instead, we compute the numerator and normalize it to sum to 1, as before. Also, note that in the final step above, we used again the d-separation properties of the Bayes net to derive that $P(o_{t+1}|s_{t+1}, o_{1:t}) = P(o_{t+1}|s_{t+1})$.

To summarize, the algorithm consists of repeatedly applying the two update rules:

Dynamics update:

$$B'(s_{t+1}) = \sum_{s_t} P(s_{t+1}|s_t)B(s_t)$$

Observation update:

$$B(s_{t+1}) = \alpha P(o_{t+1}|s_{t+1})|B'(s_{t+1})$$

where α is chosen such that $\sum_{s_{t+1}} B(s_{t+1}) = 1$.

This is sometimes called the "filtering algorithm." It is worth noting that it applies to any problem that can be posed as an HMM, not only robot localization. We will discuss other applications later. You should use this algorithm for robot localization in programming assignment 4.

3.3 Particle filtering

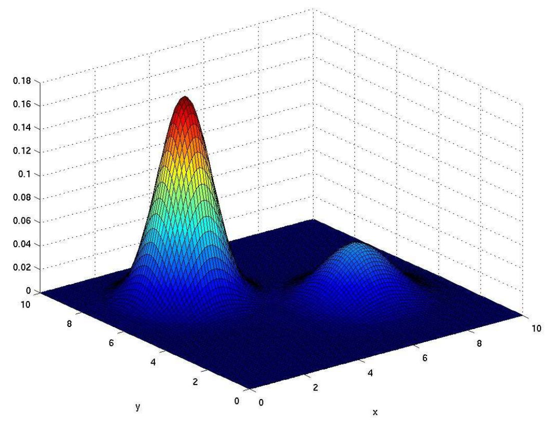
3.3.1 Sampling

So far we have always discretized the state space. This may not always be the most efficient approach. Suppose that we wanted to track the robot's orientation as well as its location. If the robot's environment is a 5 meter by 5 meter square room, we discretize its location into a square grid with 10 cm-wide cells, and we discretize its orientation into 5° bins, then it has a $50 \times 50 \times 360/5 = 180,000$ states.

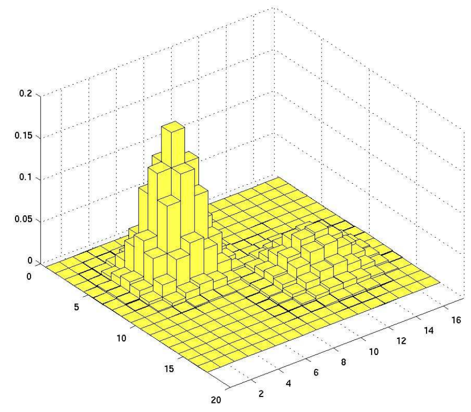
The runtime of the filtering algorithm given above is quadratic in the number of states (because of the dynamics update step), so large state spaces can be problematic. We might also observe intuitively that most of the states will have low probability, so the filtering algorithm will spend most of its time reasoning about places that the robot is not at.

Both of these problems can be addressed by using a different way of representing the state space. In reality, the robot's state is continuous, but representing a continuous distribution exactly is difficult if the distribution has a complicated shape. Instead, we can represent it approximately. Discretization is only one way of approximating a continuous distribution. We can also approximate it by representing it with a collection of samples (a set of points drawn randomly from the distribution). Intuitively, the robot is more likely to be in places with large numbers of samples, and less likely to be in places with small numbers of samples.

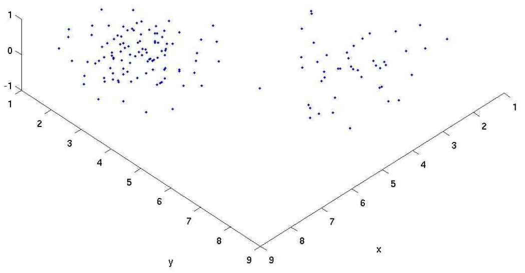
Graphically, our original distribution might look like this:



The discretized representation would look like this:



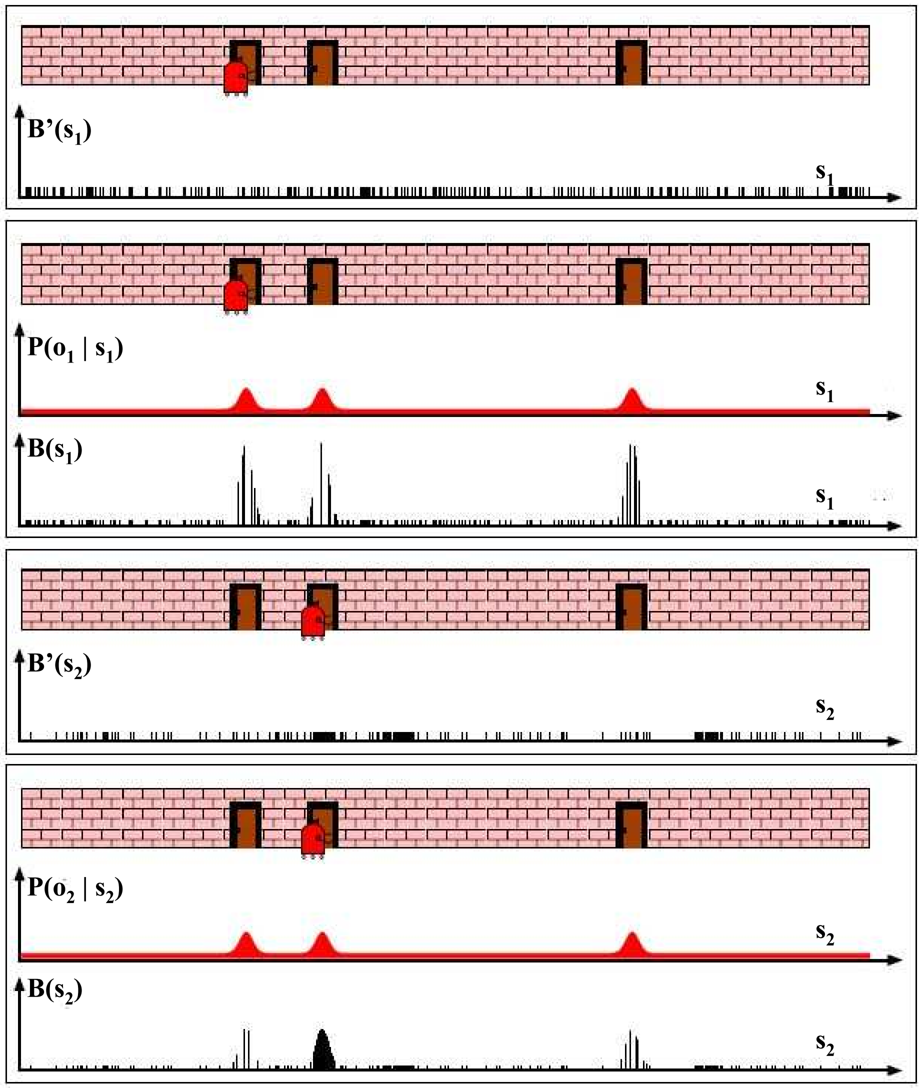
One possible collection of samples drawn from the continuous distribution looks like this. Note that the samples cluster more tightly where the original distribution had peaks:



3.3.2 The particle filtering algorithm

The **particle filtering algorithm** uses samples to represent the distribution $B(s_t)$. In the observation update, we update the set of samples by re-sampling them with replacement in such a way that samples for locations that are inconsistent with sensor readings are more likely not to be chosen, while samples for locations that are consistent with the sensors readings are more likely to be chosen, and perhaps chosen multiple times. In the dynamics update, we apply the transition model to each sample to noisily update its location.

The following figure illustrates the basic idea of the algorithm by representing samples with notches. During the observation updates, the height of the notches indicates the probability of each sample being chosen during the sampling with replacement:



(Figure courtesy of Thrun, Burgard, and Fox.)

To make this more concrete, this is the algorithm for the inductive step of the particle filter:

```
Input: $B(s_t)$ represented as a set of samples $\hat{B} = \{s^{(1)}, s^{(2)}, \ldots, s^{(m)}\}$.

For $i$=1 to $m$
    $s'^{(i)}$ = SampleDynamicsModel($s^{(i)}, a_t$ )
// $B'(s_t)$ is now represented as a set of samples $\{{s'}^{(1)}, {s'}^{(2)}, \ldots, {s'}^{(m)}\}$.
For $i$=1 to $m$
```

```
        $w^{(i)} = P(o_t|s^{(i)})$
    $\hat{B} = \emptyset$
    For $i$=1 to $m$
        Sample $i$ with probability proportional to $w$
        Add $s'^{(i)}$ to $\hat{B}$
    // Output: $B(s_{t+1})$ represented as a set of samples $\hat{B}$
```

We aren't going to fully justify this algorithm mathematically in these notes, but it does turn out to be the correct way to use samples to approximate $B(s_t)$. The intuitive justification is that states for which $w^{(i)}$ is large are more likely to be chosen, so we keep only the points for which the probability of the observation was large.

To initialize the algorithm, if our initial belief over the state space is a uniform distribution, then we would represent our initial beliefs via a set of samples $\hat{B}$ drawn from the uniform distribution over all states.

There are still two details of particle filtering that are significantly different from the discrete filtering algorithm.

The first is that the dynamics model (written SampleDynamicsModel($s^{(i)}, a_t$) above) can't just sample randomly from a discrete list of states; instead it must be capable of outputting a continuous state. Usually this just means adding random noise to a basic transformation. For example, if

$$s_t = \begin{bmatrix} x \\ y \end{bmatrix}$$

and a_t represents the action "move east," then SampleDynamicsModel($s^{(i)}, a_t$) might return

$$s_{t+1} = \begin{bmatrix} x + 0.1 \\ y \end{bmatrix} + \text{noise}$$

The other issue is that we need some way of predicting the actual location of the robot from a collection of samples. One way of doing this is to average all the points in the sample together, and return the average of all the points' positions as the estimated position of the robot. The variance (or "spread") of the sample also gives a measure of how confident the robot is in its estimate of its position. Taking a simple average as above would work fine so long as the distribution were unimodal (has only one peak). A more sophisticated algorithm might find the region of state space with the most dense collection of points, and then to average all the points in that region, and return that as the estimated position of the robot.