Machine Learning

By:
Andrew Ng

Machine Learning

By:
Andrew Ng

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

iv

Machine Learning Introduction[1]

This course provides a broad introduction to machine learning and statistical pattern recognition. Topics include: supervised learning (generative/discriminative learning, parametric/non-parametric learning, neural networks, support vector machines); unsupervised learning (clustering, dimensionality reduction, kernel methods); learning theory (bias/variance tradeoffs; VC theory; large margins); reinforcement learning and adaptive control. The course will also discuss recent applications of machine learning, such as to robotic control, data mining, autonomous navigation, bioinformatics, speech recognition, and text and web data processing.Students are expected to have the following background:Prerequisites: - Knowledge of basic computer science principles and skills, at a level sufficient to write a reasonably non-trivial computer program.- Familiarity with the basic probability theory. (Stat 116 is sufficient but not necessary.)- Familiarity with the basic linear algebra (any one of Math 51, Math 103, Math 113, or CS 205 would be much more than necessary.)

1

Ng's research is in the areas of machine learning and artificial intelligence. He leads the STAIR (STanford Artificial Intelligence Robot) project, whose goal is to develop a home assistant robot

---

[1]This content is available online at <http://cnx.org/content/m45891/1.2/>.

that can perform tasks such as tidy up a room, load/unload a dishwasher, fetch and deliver items, and prepare meals using a kitchen. Since its birth in 1956, the AI dream has been to build systems that exhibit "broad spectrum" intelligence. However, AI has since splintered into many different subfields, such as machine learning, vision, navigation, reasoning, planning, and natural language processing. To realize its vision of a home assistant robot, STAIR will unify into a single platform tools drawn from all of these AI subfields. This is in distinct contrast to the 30-year-old trend of working on fragmented AI sub-fields, so that STAIR is also a unique vehicle for driving forward research towards true, integrated AI.

Ng also works on machine learning algorithms for robotic control, in which rather than relying on months of human hand-engineering to design a controller, a robot instead learns automatically how best to control itself. Using this approach, Ng's group has developed by far the most advanced autonomous helicopter controller, that is capable of flying spectacular aerobatic maneuvers that even experienced human pilots often find extremely difficult to execute. As part of this work, Ng's group also developed algorithms that can take a single image,and turn the picture into a 3-D model that one can fly-through and see from different angles.

Machine Learning Syllabus[2]

Course Description

This course provides a broad introduction to machine learning and statistical pattern recognition. Topics include: supervised learning (generative/discriminative learning, parametric/non-parametric learning, neural networks, support vector machines); unsupervised learning (clustering, dimensionality reduction, kernel methods); learning theory (bias/variance tradeoffs; VC theory; large margins); reinforcement learning and adaptive control. The course will also discuss recent applications of machine learning, such as to robotic control, data mining, autonomous navigation, bioinformatics, speech recognition, and text and web data processing.

Prerequisites

Students are expected to have the following background:

- Knowledge of basic computer science principles and skills, at a level sufficient to write a reasonably non-trivial computer program.
- Familiarity with the basic probability theory. (Stat 116 is sufficient but not necessary.)
- Familiarity with the basic linear algebra (any one of Math 51, Math 103, Math 113, or CS 205 would be much more than necessary.)

Course Materials

There is no required text for this course. Notes will be posted periodically on the course web site. The following books are recommended as optional reading:

- Christopher Bishop, Pattern Recognition and Machine Learning. Springer, 2006.
- Richard Duda, Peter Hart and David Stork, Pattern Classification, 2nd ed. John Wiley & Sons, 2001.
- Tom Mitchell, Machine Learning. McGraw-Hill, 1997. Richard Sutton and Andrew Barto, Reinforcement Learning: An introduction. MIT Press, 1998

Homeworks and Grading

There will be four written homeworks, one midterm, and one major open-ended term project. The homeworks will contain written questions and questions that require some Matlab programming. In the term project, you will investigate some interesting aspect of machine learning or apply machine learning to a problem that interests you. We try very hard to make questions unambiguous, but some ambiguities may remain. Ask if confused or state your assumptions explicitly. Reasonable assumptions will be accepted in case of ambiguous questions.

---

[2]This content is available online at <http://cnx.org/content/m45958/1.2/>.

4

A note on the honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

Late homeworks: Recognizing that students may face unusual circumstances and require some flexibility in the course of the quarter, each student will have a total of seven free late (calendar) days to use as s/he sees fit. Once these late days are exhausted, any homework turned in late will be penalized 20% per late day. However, no homework will be accepted more than four days after its due date, and late days cannot be used for the final project writeup. Each 24 hours or part thereof that a homework is late uses up one full late day.

Course grades will be based 40% on homeworks (10% each), 20% on the midterm, and 40% on the major term project. Up to 3% extra credit may be awarded for class participation.

Sections

To review material from the prerequisites or to supplement the lecture material, there will occasionally be extra discussion sections held on Friday. An announcement will be made whenever one of these sections is held. Attendance at these sections is optional.

Chapter 1

Lectures

1.1 Machine Learning Lecture 1[1]

One Small Step for (a) Man

Figure 1.1

Instructor (Andrew Ng): Okay. Good morning. Welcome to CS229, the machine learning class. So what I wanna do today is just spend a little time going over the logistics of the class, and then we'll start to talk a bit about machine learning.

By way of introduction, my name's Andrew Ng and I'll be instructor for this class. And so I personally work in machine learning, and I've worked on it for about 15 years now, and I actually think that machine learning is the most exciting field of all the computer sciences. So I'm actually always excited about teaching this class. Sometimes I actually think that machine learning is not only the most exciting thing in computer science, but the most exciting thing in all of human endeavor, so maybe a little bias there.

I also want to introduce the TAs, who are all graduate students doing research in or related to the machine learning and all aspects of machine learning. Paul Baumstarck works in machine learning and computer vision. Catie Chang is actually a neuroscientist who applies machine learning algorithms to try to understand the human brain. Tom Do is another PhD student, works in computational biology and in sort of the basic fundamentals of human learning. Zico Kolter is the head TA — he's head TA two years in a row now — works in machine learning and applies them to a bunch of robots. And Daniel Ramage is — I guess he's not here — Daniel applies learning algorithms to problems in natural language processing.

So you'll get to know the TAs and me much better throughout this quarter, but just from the sorts of things the TA's do, I hope you can already tell that machine learning is a highly interdisciplinary topic in which just the TAs find learning algorithms to problems in computer vision and biology and robots and language. And machine learning is one of those things that has and is having a large impact on many applications.

So just in my own daily work, I actually frequently end up talking to people like helicopter pilots to biologists to people in computer systems or databases to economists and sort of also an unending stream of people from industry coming to Stanford interested in applying machine learning methods to their own problems.

---

[1]This content is available online at <http://cnx.org/content/m45863/1.3/>.

So yeah, this is fun. A couple of weeks ago, a student actually forwarded to me an article in "Computer World" about the 12 IT skills that employers can't say no to. So it's about sort of the 12 most desirable skills in all of IT and all of information technology, and topping the list was actually machine learning. So I think this is a good time to be learning this stuff and learning algorithms and having a large impact on many segments of science and industry.

I'm actually curious about something. Learning algorithms is one of the things that touches many areas of science and industries, and I'm just kind of curious. How many people here are computer science majors, are in the computer science department? Okay. About half of you. How many people are from EE? Oh, okay, maybe about a fifth. How many biologers are there here? Wow, just a few, not many. I'm surprised. Anyone from statistics? Okay, a few. So where are the rest of you from?

Student : iCME.

Instructor (Andrew Ng) : Say again?

Student : iCME.

Instructor (Andrew Ng) : iCME. Cool.

Student : [Inaudible].

Instructor (Andrew Ng) : Civi and what else?

Student : [Inaudible]

Instructor (Andrew Ng) : Synthesis, [inaudible] systems. Yeah, cool.

Student : Chemi.

Instructor (Andrew Ng) : Chemi. Cool.

Student : [Inaudible].

Instructor (Andrew Ng) : Aero/astro. Yes, right. Yeah, okay, cool. Anyone else?

Student : [Inaudible].

Instructor (Andrew Ng) : Pardon? MSNE. All right. Cool. Yeah.

Student : [Inaudible].

Instructor (Andrew Ng) : Pardon?

Student : [Inaudible].

Instructor (Andrew Ng) : Endo —

Student : [Inaudible].

Instructor (Andrew Ng) : Oh, I see, industry. Okay. Cool. Great, great. So as you can tell from a cross-section of this class, I think we're a very diverse audience in this room, and that's one of the things that makes this class fun to teach and fun to be in, I think.

So in this class, we've tried to convey to you a broad set of principles and tools that will be useful for doing many, many things. And every time I teach this class, I can actually very confidently say that after December, no matter what you're going to do after this December when you've sort of completed this class, you'll find the things you learn in this class very useful, and these things will be useful pretty much no matter what you end up doing later in your life.

So I have more logistics to go over later, but let's say a few more words about machine learning. I feel that machine learning grew out of early work in AI, early work in artificial intelligence. And over the last — I wanna say last 15 or last 20 years or so, it's been viewed as a sort of growing new capability for computers. And in particular, it turns out that there are many programs or there are many applications that you can't program by hand.

For example, if you want to get a computer to read handwritten characters, to read sort of handwritten digits, that actually turns out to be amazingly difficult to write a piece of software to take this input, an image of something that I wrote and to figure out just what it is, to translate my cursive handwriting into — to extract the characters I wrote out in longhand. And other things: One thing that my students and I do is autonomous flight. It turns out to be extremely difficult to sit down and write a program to fly a helicopter.

But in contrast, if you want to do things like to get software to fly a helicopter or have software recognize handwritten digits, one very successful approach is to use a learning algorithm and have a computer learn

by itself how to, say, recognize your handwriting. And in fact, handwritten digit recognition, this is pretty much the only approach that works well. It uses applications that are hard to program by hand.

Learning algorithms has also made I guess significant inroads in what's sometimes called database mining. So, for example, with the growth of IT and computers, increasingly many hospitals are keeping around medical records of what sort of patients, what problems they had, what their prognoses was, what the outcome was. And taking all of these medical records, which started to be digitized only about maybe 15 years, applying learning algorithms to them can turn raw medical records into what I might loosely call medical knowledge in which we start to detect trends in medical practice and even start to alter medical practice as a result of medical knowledge that's derived by applying learning algorithms to the sorts of medical records that hospitals have just been building over the last 15, 20 years in an electronic format.

Turns out that most of you probably use learning algorithms — I don't know — I think half a dozen times a day or maybe a dozen times a day or more, and often without knowing it. So, for example, every time you send mail via the US Postal System, turns out there's an algorithm that tries to automatically read the zip code you wrote on your envelope, and that's done by a learning algorithm. So every time you send US mail, you are using a learning algorithm, perhaps without even being aware of it.

Similarly, every time you write a check, I actually don't know the number for this, but a significant fraction of checks that you write are processed by a learning algorithm that's learned to read the digits, so the dollar amount that you wrote down on your check. So every time you write a check, there's another learning algorithm that you're probably using without even being aware of it.

If you use a credit card, or I know at least one phone company was doing this, and lots of companies like eBay as well that do electronic transactions, there's a good chance that there's a learning algorithm in the background trying to figure out if, say, your credit card's been stolen or if someone's engaging in a fraudulent transaction.

If you use a website like Amazon or Netflix that will often recommend books for you to buy or movies for you to rent or whatever, these are other examples of learning algorithms that have learned what sorts of things you like to buy or what sorts of movies you like to watch and can therefore give customized recommendations to you.

Just about a week ago, I had my car serviced, and even there, my car mechanic was trying to explain to me some learning algorithm in the innards of my car that's sort of doing its best to optimize my driving performance for fuel efficiency or something.

So, see, most of us use learning algorithms half a dozen, a dozen, maybe dozens of times without even knowing it.

And of course, learning algorithms are also doing things like giving us a growing understanding of the human genome. So if someday we ever find a cure for cancer, I bet learning algorithms will have had a large role in that. That's sort of the thing that Tom works on, yes?

So in teaching this class, I sort of have three goals. One of them is just to I hope convey some of my own excitement about machine learning to you.

The second goal is by the end of this class, I hope all of you will be able to apply state-of-the-art machine learning algorithms to whatever problems you're interested in. And if you ever need to build a system for reading zip codes, you'll know how to do that by the end of this class.

And lastly, by the end of this class, I realize that only a subset of you are interested in doing research in machine learning, but by the conclusion of this class, I hope that all of you will actually be well qualified to start doing research in machine learning, okay?

So let's say a few words about logistics. The prerequisites of this class are written on one of the handouts, are as follows: In this class, I'm going to assume that all of you have sort of basic knowledge of computer science and knowledge of the basic computer skills and principles. So I assume all of you know what big?O notation, that all of you know about sort of data structures like queues, stacks, binary trees, and that all of you know enough programming skills to, like, write a simple computer program. And it turns out that most of this class will not be very programming intensive, although we will do some programming, mostly in either MATLAB or Octave. I'll say a bit more about that later.

I also assume familiarity with basic probability and statistics. So most undergraduate statistics class, like

Stat 116 taught here at Stanford, will be more than enough. I'm gonna assume all of you know what random variables are, that all of you know what expectation is, what a variance or a random variable is. And in case of some of you, it's been a while since you've seen some of this material. At some of the discussion sections, we'll actually go over some of the prerequisites, sort of as a refresher course under prerequisite class. I'll say a bit more about that later as well.

Lastly, I also assume familiarity with basic linear algebra. And again, most undergraduate linear algebra courses are more than enough. So if you've taken courses like Math 51, 103, Math 113 or CS205 at Stanford, that would be more than enough. Basically, I'm gonna assume that all of you know what matrixes and vectors are, that you know how to multiply matrices and vectors and multiply matrix and matrices, that you know what a matrix inverse is. If you know what an eigenvector of a matrix is, that'd be even better. But if you don't quite know or if you're not quite sure, that's fine, too. We'll go over it in the review sections.

So there are a couple more logistical things I should deal with in this class. One is that, as most of you know, CS229 is a televised class. And in fact, I guess many of you are probably watching this at home on TV, so I'm gonna say hi to our home viewers.

So earlier this year, I approached SCPD, which televises these classes, about trying to make a small number of Stanford classes publicly available or posting the videos on the web. And so this year, Stanford is actually starting a small pilot program in which we'll post videos of a small number of classes online, so on the Internet in a way that makes it publicly accessible to everyone. I'm very excited about that because machine learning in school, let's get the word out there.

One of the consequences of this is that — let's see — so videos or pictures of the students in this classroom will not be posted online, so your images — so don't worry about being by seeing your own face appear on YouTube one day. But the microphones may pick up your voices, so I guess the consequence of that is that because microphones may pick up your voices, no matter how irritated you are at me, don't yell out swear words in the middle of class, but because there won't be video you can safely sit there and make faces at me, and that won't show, okay?

Let's see. I also handed out this — there were two handouts I hope most of you have, course information handout. So let me just say a few words about parts of these. On the third page, there's a section that says Online Resources.

Oh, okay. Louder? Actually, could you turn up the volume? Testing. Is this better? Testing, testing. Okay, cool. Thanks.

So all right, online resources. The class has a home page, so it's in on the handouts. I won't write on the chalkboard — http://cs229.stanford.edu. And so when there are homework assignments or things like that, we usually won't sort of — in the mission of saving trees, we will usually not give out many handouts in class. So homework assignments, homework solutions will be posted online at the course home page.

As far as this class, I've also written, and I guess I've also revised every year a set of fairly detailed lecture notes that cover the technical content of this class. And so if you visit the course homepage, you'll also find the detailed lecture notes that go over in detail all the math and equations and so on that I'll be doing in class.

There's also a newsgroup, su.class.cs229, also written on the handout. This is a newsgroup that's sort of a forum for people in the class to get to know each other and have whatever discussions you want to have amongst yourselves. So the class newsgroup will not be monitored by the TAs and me. But this is a place for you to form study groups or find project partners or discuss homework problems and so on, and it's not monitored by the TAs and me. So feel free to talk trash about this class there.

If you want to contact the teaching staff, please use the email address written down here, cs229-qa@cs.stanford.edu. This goes to an account that's read by all the TAs and me. So rather than sending us email individually, if you send email to this account, it will actually let us get back to you maximally quickly with answers to your questions.

If you're asking questions about homework problems, please say in the subject line which assignment and which question the email refers to, since that will also help us to route your question to the appropriate TA or to me appropriately and get the response back to you quickly.

Let's see. Skipping ahead — let's see — for homework, one midterm, one open and term project. Notice

on the honor code. So one thing that I think will help you to succeed and do well in this class and even help you to enjoy this class more is if you form a study group.

So start looking around where you're sitting now or at the end of class today, mingle a little bit and get to know your classmates. I strongly encourage you to form study groups and sort of have a group of people to study with and have a group of your fellow students to talk over these concepts with. You can also post on the class newsgroup if you want to use that to try to form a study group.

But some of the problems sets in this class are reasonably difficult. People that have taken the class before may tell you they were very difficult. And just I bet it would be more fun for you, and you'd probably have a better learning experience if you form a study group of people to work with. So I definitely encourage you to do that.

And just to say a word on the honor code, which is I definitely encourage you to form a study group and work together, discuss homework problems together. But if you discuss homework problems with other students, then I'll ask you to sort of go home and write down your own solutions independently without referring to notes that were taken in any of your joint study sessions.

So in other words, when you turn in a homework problem, what you turn in should be something that was reconstructed independently by yourself and without referring to notes that you took during your study sessions with other people, okay? And obviously, showing your solutions to others or copying other solutions directly is right out.

We occasionally also reuse problem set questions from previous years so that the problems are a bit more debugged and work more smoothly. And as a result of that, I also ask you not to look at solutions from previous years, and this includes both sort of official solutions that we've given out to previous generations of this class and previous solutions that people that have taken this class in previous years may have written out by themselves, okay?

Sadly, in this class, there are usually — sadly, in previous years, there have often been a few honor code violations in this class. And last year, I think I prosecuted five honor code violations, which I think is a ridiculously large number. And so just don't work without solutions, and hopefully there'll be zero honor code violations this year. I'd love for that to happen.

The section here on the late homework policy if you ever want to hand in a homework late, I'll leave you to read that yourself.

We also have a midterm, which is scheduled for Thursday, 8th of November at 6:00 p.m., so please keep that evening free.

And let's see. And one more administrative thing I wanted to say is about the class project. So part of the goal of this class is to leave you well equipped to apply machine learning algorithms to a problem or to do research in machine learning. And so as part of this class, I'll ask you to execute a small research project sort of as a small term project.

And what most students do for this is either apply machine learning to a problem that you find interesting or investigate some aspect of machine learning. So to those of you that are either already doing research or to those of you who are in industry, you're taking this from a company, one fantastic sort of way to do a class project would be if you apply machine learning algorithms to a problem that you're interested in, to a problem that you're already working on, whether it be a science research problem or sort of a problem in industry where you're trying to get a system to work using a learning algorithm.

To those of you that are not currently doing research, one great way to do a project would be if you apply learning algorithms to just pick a problem that you care about. Pick a problem that you find interesting, and apply learning algorithms to that and play with the ideas and see what happens.

And let's see. Oh, and the goal of the project should really be for you to do a publishable piece of research in machine learning, okay?

And if you go to the course website, you'll actually find a list of the projects that students had done last year. And so I'm holding the list in my hand. You can go home later and take a look at it online.

But reading down this list, I see that last year, there were students that applied learning algorithms to control a snake robot. There was a few projects on improving learning algorithms. There's a project on flying autonomous aircraft. There was a project actually done by our TA Paul on improving computer vision

algorithms using machine learning.

There are a couple of projects on Netflix rankings using learning algorithms; a few medical robots; ones on segmenting [inaudible] to segmenting pieces of the body using learning algorithms; one on musical instrument detection; another on irony sequence alignment; and a few algorithms on understanding the brain neuroscience, actually quite a few projects on neuroscience; a couple of projects on undescending fMRI data on brain scans, and so on; another project on market makings, the financial trading. There was an interesting project on trying to use learning algorithms to decide what is it that makes a person's face physically attractive. There's a learning algorithm on optical illusions, and so on.

And it goes on, so lots of fun projects. And take a look, then come up with your own ideas. But whatever you find cool and interesting, I hope you'll be able to make machine learning a project out of it. Yeah, question?

Student : Are these group projects?

Instructor (Andrew Ng): Oh, yes, thank you.

Student : So how many people can be in a group?

Instructor (Andrew Ng): Right. So projects can be done in groups of up to three people. So as part of forming study groups, later today as you get to know your classmates, I definitely also encourage you to grab two other people and form a group of up to three people for your project, okay? And just start brainstorming ideas for now amongst yourselves. You can also come and talk to me or the TAs if you want to brainstorm ideas with us.

Okay. So one more organizational question. I'm curious, how many of you know MATLAB? Wow, cool, quite a lot. Okay. So as part of the — actually how many of you know Octave or have used Octave? Oh, okay, much smaller number.

So as part of this class, especially in the homeworks, we'll ask you to implement a few programs, a few machine learning algorithms as part of the homeworks. And most of those homeworks will be done in either MATLAB or in Octave, which is sort of — I know some people call it a free version of MATLAB, which it sort of is, sort of isn't.

So I guess for those of you that haven't seen MATLAB before, and I know most of you have, MATLAB is I guess part of the programming language that makes it very easy to write codes using matrices, to write code for numerical routines, to move data around, to plot data. And it's sort of an extremely easy to learn tool to use for implementing a lot of learning algorithms.

And in case some of you want to work on your own home computer or something if you don't have a MATLAB license, for the purposes of this class, there's also — [inaudible] write that down [inaudible] MATLAB — there's also a software package called Octave that you can download for free off the Internet. And it has somewhat fewer features than MATLAB, but it's free, and for the purposes of this class, it will work for just about everything.

So actually I, well, so yeah, just a side comment for those of you that haven't seen MATLAB before I guess, once a colleague of mine at a different university, not at Stanford, actually teaches another machine learning course. He's taught it for many years. So one day, he was in his office, and an old student of his from, like, ten years ago came into his office and he said, "Oh, professor, professor, thank you so much for your machine learning class. I learned so much from it. There's this stuff that I learned in your class, and I now use every day. And it's helped me make lots of money, and here's a picture of my big house."

So my friend was very excited. He said, "Wow. That's great. I'm glad to hear this machine learning stuff was actually useful. So what was it that you learned? Was it logistic regression? Was it the PCA? Was it the data networks? What was it that you learned that was so helpful?" And the student said, "Oh, it was the MATLAB."

So for those of you that don't know MATLAB yet, I hope you do learn it. It's not hard, and we'll actually have a short MATLAB tutorial in one of the discussion sections for those of you that don't know it.

Okay. The very last piece of logistical thing is the discussion sections. So discussion sections will be taught by the TAs, and attendance at discussion sections is optional, although they'll also be recorded and televised. And we'll use the discussion sections mainly for two things. For the next two or three weeks, we'll use the discussion sections to go over the prerequisites to this class or if some of you haven't seen probability

or statistics for a while or maybe algebra, we'll go over those in the discussion sections as a refresher for those of you that want one.

Later in this quarter, we'll also use the discussion sections to go over extensions for the material that I'm teaching in the main lectures. So machine learning is a huge field, and there are a few extensions that we really want to teach but didn't have time in the main lectures for.

So later this quarter, we'll use the discussion sections to talk about things like convex optimization, to talk a little bit about hidden Markov models, which is a type of machine learning algorithm for modeling time series and a few other things, so extensions to the materials that I'll be covering in the main lectures. And attendance at the discussion sections is optional, okay?

So that was all I had from logistics. Before we move on to start talking a bit about machine learning, let me check what questions you have. Yeah?

Student : [Inaudible] R or something like that?

Instructor (Andrew Ng) : Oh, yeah, let's see, right. So our policy has been that you're welcome to use R, but I would strongly advise against it, mainly because in the last problem set, we actually supply some code that will run in Octave but that would be somewhat painful for you to translate into R yourself. So for your other assignments, if you wanna submit a solution in R, that's fine. But I think MATLAB is actually totally worth learning. I know R and MATLAB, and I personally end up using MATLAB quite a bit more often for various reasons. Yeah?

Student : For the [inaudible] project [inaudible]?

Instructor (Andrew Ng) : So for the term project, you're welcome to do it in smaller groups of three, or you're welcome to do it by yourself or in groups of two. Grading is the same regardless of the group size, so with a larger group, you probably — I recommend trying to form a team, but it's actually totally fine to do it in a smaller group if you want.

Student : [Inaudible] what language [inaudible]?

Instructor (Andrew Ng): So let's see. There is no C programming in this class other than any that you may choose to do yourself in your project. So all the homeworks can be done in MATLAB or Octave, and let's see. And I guess the program prerequisites is more the ability to understand big?O notation and knowledge of what a data structure, like a linked list or a queue or binary treatments, more so than your knowledge of C or Java specifically. Yeah?

Student : Looking at the end semester project, I mean, what exactly will you be testing over there? [Inaudible]?

Instructor (Andrew Ng) : Of the project?

Student : Yeah.

Instructor (Andrew Ng) : Yeah, let me answer that later. In a couple of weeks, I shall give out a handout with guidelines for the project. But for now, we should think of the goal as being to do a cool piece of machine learning work that will let you experience the joys of machine learning firsthand and really try to think about doing a publishable piece of work.

So many students will try to build a cool machine learning application. That's probably the most common project. Some students will try to improve state-of-the-art machine learning. Some of those projects are also very successful. It's a little bit harder to do. And there's also a smaller minority of students that will sometimes try to prove — develop the theory of machine learning further or try to prove theorems about machine learning. So they're usually great projects of all of those types with applications and machine learning being the most common. Anything else? Okay, cool.

So that was it for logistics. Let's talk about learning algorithms. So can I have the laptop display, please, or the projector? Actually, could you lower the big screen? Cool. This is amazing customer service. Thank you. I see. Okay, cool. Okay. No, that's fine. I see. Okay. That's cool. Thanks. Okay.

Big screen isn't working today, but I hope you can read things on the smaller screens out there. Actually, [inaudible] I think this room just got a new projector that — someone sent you an excited email — was it just on Friday? — saying we just got a new projector and they said 4,000-to-1 something or other brightness ratio. I don't know. Someone was very excited about the new projector in this room, but I guess we'll see that in operation on Wednesday.

So start by talking about what machine learning is. What is machine learning? Actually, can you read the text out there? Raise your hand if the text on the small screens is legible. Oh, okay, cool, mostly legible. Okay. So I'll just read it out.

So what is machine learning? Way back in about 1959, Arthur Samuel defined machine learning informally as the [inaudible] that gives computers to learn — [inaudible] that gives computers the ability to learn without being explicitly programmed. So Arthur Samuel, so way back in the history of machine learning, actually did something very cool, which was he wrote a checkers program, which would play games of checkers against itself.

And so because a computer can play thousands of games against itself relatively quickly, Arthur Samuel had his program play thousands of games against itself, and over time it would start to learn to recognize patterns which led to wins and patterns which led to losses. So over time it learned things like that, "Gee, if I get a lot of pieces taken by the opponent, then I'm more likely to lose than win," or, "Gee, if I get my pieces into a certain position, then I'm especially likely to win rather than lose."

And so over time, Arthur Samuel had a checkers program that would actually learn to play checkers by learning what are the sort of board positions that tend to be associated with wins and what are the board positions that tend to be associated with losses. And way back around 1959, the amazing thing about this was that his program actually learned to play checkers much better than Arthur Samuel himself could.

So even today, there are some people that say, well, computers can't do anything that they're not explicitly programmed to. And Arthur Samuel's checkers program was maybe the first I think really convincing refutation of this claim. Namely, Arthur Samuel managed to write a checkers program that could play checkers much better than he personally could, and this is an instance of maybe computers learning to do things that they were not programmed explicitly to do.

Here's a more recent, a more modern, more formal definition of machine learning due to Tom Mitchell, who says that a well-posed learning problem is defined as follows: He says that a computer program is set to learn from an 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. Okay. So not only is it a definition, it even rhymes.

So, for example, in the case of checkers, the experience E that a program has would be the experience of playing lots of games of checkers against itself, say. The task T is the task of playing checkers, and the performance measure P will be something like the fraction of games it wins against a certain set of human opponents. And by this definition, we'll say that Arthur Samuel's checkers program has learned to play checkers, okay?

So as an overview of what we're going to do in this class, this class is sort of organized into four major sections. We're gonna talk about four major topics in this class, the first of which is supervised learning. So let me give you an example of that.

So suppose you collect a data set of housing prices. And one of the TAs, Dan Ramage, actually collected a data set for me last week to use in the example later. But suppose that you go to collect statistics about how much houses cost in a certain geographic area. And Dan, the TA, collected data from housing prices in Portland, Oregon. So what you can do is let's say plot the square footage of the house against the list price of the house, right, so you collect data on a bunch of houses. And let's say you get a data set like this with houses of different sizes that are listed for different amounts of money.

Now, let's say that I'm trying to sell a house in the same area as Portland, Oregon as where the data comes from. Let's say I have a house that's this size in square footage, and I want an algorithm to tell me about how much should I expect my house to sell for. So there are lots of ways to do this, and some of you may have seen elements of what I'm about to say before.

So one thing you could do is look at this data and maybe put a straight line to it. And then if this is my house, you may then look at the straight line and predict that my house is gonna go for about that much money, right? There are other decisions that we can make, which we'll talk about later, which is, well, what if I don't wanna put a straight line? Maybe I should put a quadratic function to it. Maybe that fits the data a little bit better. You notice if you do that, the price of my house goes up a bit, so that'd be nice.

And this sort of learning problem of learning to predict housing prices is an example of what's called a supervised learning problem. And the reason that it's called supervised learning is because we're providing

the algorithm a data set of a bunch of square footages, a bunch of housing sizes, and as well as sort of the right answer of what the actual prices of a number of houses were, right?

So we call this supervised learning because we're supervising the algorithm or, in other words, we're giving the algorithm the, quote, right answer for a number of houses. And then we want the algorithm to learn the association between the inputs and the outputs and to sort of give us more of the right answers, okay?

It turns out this specific example that I drew here is an example of something called a regression problem. And the term regression sort of refers to the fact that the variable you're trying to predict is a continuous value and price.

There's another class of supervised learning problems which we'll talk about, which are classification problems. And so, in a classification problem, the variable you're trying to predict is discreet rather than continuous. So as one specific example — so actually a standard data set you can download online [inaudible] that lots of machine learning people have played with. Let's say you collect a data set on breast cancer tumors, and you want to learn the algorithm to predict whether or not a certain tumor is malignant. Malignant is the opposite of benign, right, so malignancy is a sort of harmful, bad tumor. So we collect some number of features, some number of properties of these tumors, and for the sake of sort of having a simple [inaudible] explanation, let's just say that we're going to look at the size of the tumor and depending on the size of the tumor, we'll try to figure out whether or not the tumor is malignant or benign.

So the tumor is either malignant or benign, and so the variable in the Y axis is either zero or 1, and so your data set may look something like that, right? And that's 1 and that's zero, okay? And so this is an example of a classification problem where the variable you're trying to predict is a discreet value. It's either zero or 1.

And in fact, more generally, there will be many learning problems where we'll have more than one input variable, more than one input feature and use more than one variable to try to predict, say, whether a tumor is malignant or benign. So, for example, continuing with this, you may instead have a data set that looks like this. I'm gonna part this data set in a slightly different way now. And I'm making this data set look much cleaner than it really is in reality for illustration, okay?

For example, maybe the crosses indicate malignant tumors and the "O"s may indicate benign tumors. And so you may have a data set comprising patients of different ages and who have different tumor sizes and where a cross indicates a malignant tumor, and an "O" indicates a benign tumor. And you may want an algorithm to learn to predict, given a new patient, whether their tumor is malignant or benign.

So, for example, what a learning algorithm may do is maybe come in and decide that a straight line like that separates the two classes of tumors really well, and so if you have a new patient who's age and tumor size fall over there, then the algorithm may predict that the tumor is benign rather than malignant, okay? So this is just another example of another supervised learning problem and another classification problem.

And so it turns out that one of the issues we'll talk about later in this class is in this specific example, we're going to try to predict whether a tumor is malignant or benign based on two features or based on two inputs, namely the age of the patient and the tumor size. It turns out that when you look at a real data set, you find that learning algorithms often use other sets of features. In the breast cancer data example, you also use properties of the tumors, like clump thickness, uniformity of cell size, uniformity of cell shape, [inaudible] adhesion and so on, so various other medical properties.

And one of the most interesting things we'll talk about later this quarter is what if your data doesn't lie in a two-dimensional or three-dimensional or sort of even a finite dimensional space, but is it possible — what if your data actually lies in an infinite dimensional space? Our plots here are two-dimensional space. I can't plot you an infinite dimensional space, right? And so it turns out that one of the most successful classes of machine learning algorithms — some may call support vector machines — actually takes data and maps data to an infinite dimensional space and then does classification using not two features like I've done here, but an infinite number of features.

And that will actually be one of the most fascinating things we talk about when we go deeply into classification algorithms. And it's actually an interesting question, right, so think about how do you even represent an infinite dimensional vector in computer memory? You don't have an infinite amount of computers. How

do you even represent a point that lies in an infinite dimensional space? We'll talk about that when we get to support vector machines, okay?

So let's see. So that was supervised learning. The second of the four major topics of this class will be learning theory. So I have a friend who teaches math at a different university, not at Stanford, and when you talk to him about his work and what he's really out to do, this friend of mine will — he's a math professor, right? — this friend of mine will sort of get the look of wonder in his eyes, and he'll tell you about how in his mathematical work, he feels like he's discovering truth and beauty in the universe. And he says it in sort of a really touching, sincere way, and then he has this — you can see it in his eyes — he has this deep appreciation of the truth and beauty in the universe as revealed to him by the math he does.

In this class, I'm not gonna do any truth and beauty. In this class, I'm gonna talk about learning theory to try to convey to you an understanding of how and why learning algorithms work so that we can apply these learning algorithms as effectively as possible.

So, for example, it turns out you can prove surprisingly deep theorems on when you can guarantee that a learning algorithm will work, all right? So think about a learning algorithm for reading zip codes. When can you prove a theorem guaranteeing that a learning algorithm will be at least 99.9 percent accurate on reading zip codes? This is actually somewhat surprising. We actually prove theorems showing when you can expect that to hold.

We'll also sort of delve into learning theory to try to understand what algorithms can approximate different functions well and also try to understand things like how much training data do you need? So how many examples of houses do I need in order for your learning algorithm to recognize the pattern between the square footage of a house and its housing price? And this will help us answer questions like if you're trying to design a learning algorithm, should you be spending more time collecting more data or is it a case that you already have enough data; it would be a waste of time to try to collect more. Okay?

So I think learning algorithms are a very powerful tool that as I walk around sort of industry in Silicon Valley or as I work with various businesses in CS and outside CS, I find that there's often a huge difference between how well someone who really understands this stuff can apply a learning algorithm versus someone who sort of gets it but sort of doesn't.

The analogy I like to think of is imagine you were going to a carpentry school instead of a machine learning class, right? If you go to a carpentry school, they can give you the tools of carpentry. They'll give you a hammer, a bunch of nails, a screwdriver or whatever. But a master carpenter will be able to use those tools far better than most of us in this room. I know a carpenter can do things with a hammer and nail that I couldn't possibly. And it's actually a little bit like that in machine learning, too. One thing that's sadly not taught in many courses on machine learning is how to take the tools of machine learning and really, really apply them well.

So in the same way, so the tools of machine learning are I wanna say quite a bit more advanced than the tools of carpentry. Maybe a carpenter will disagree. But a large part of this class will be just giving you the raw tools of machine learning, just the algorithms and so on. But what I plan to do throughout this entire quarter, not just in the segment of learning theory, but actually as a theme running through everything I do this quarter, will be to try to convey to you the skills to really take the learning algorithm ideas and really to get them to work on a problem.

It's sort of hard for me to stand here and say how big a deal that is, but when I walk around companies in Silicon Valley, it's completely not uncommon for me to see someone using some machine learning algorithm and then explain to me what they've been doing for the last six months, and I go, oh, gee, it should have been obvious from the start that the last six months, you've been wasting your time, right?

And so my goal in this class, running through the entire quarter, not just on learning theory, is actually not only to give you the tools of machine learning, but to teach you how to use them well. And I've noticed this is something that really not many other classes teach. And this is something I'm really convinced is a huge deal, and so by the end of this class, I hope all of you will be master carpenters. I hope all of you will be really good at applying these learning algorithms and getting them to work amazingly well in many problems. Okay?

Let's see. So [inaudible] the board. After learning theory, there's another class of learning algorithms

that I then want to teach you about, and that's unsupervised learning. So you recall, right, a little earlier I drew an example like this, right, where you have a couple of features, a couple of input variables and sort of malignant tumors and benign tumors or whatever. And that was an example of a supervised learning problem because the data you have gives you the right answer for each of your patients. The data tells you this patient has a malignant tumor; this patient has a benign tumor. So it had the right answers, and you wanted the algorithm to just produce more of the same.

In contrast, in an unsupervised learning problem, this is the sort of data you get, okay? Where speaking loosely, you're given a data set, and I'm not gonna tell you what the right answer is on any of your data. I'm just gonna give you a data set and I'm gonna say, "Would you please find interesting structure in this data set?" So that's the unsupervised learning problem where you're sort of not given the right answer for everything.

So, for example, an algorithm may find structure in the data in the form of the data being partitioned into two clusters, or clustering is sort of one example of an unsupervised learning problem.

So I hope you can see this. It turns out that these sort of unsupervised learning algorithms are also used in many problems. This is a screen shot — this is a picture I got from Sue Emvee, who's a PhD student here, who is applying unsupervised learning algorithms to try to understand gene data, so is trying to look at genes as individuals and group them into clusters based on properties of what genes they respond to — based on properties of how the genes respond to different experiments.

Another interesting application of [inaudible] sorts of clustering algorithms is actually image processing, this which I got from Steve Gules, who's another PhD student. It turns out what you can do is if you give this sort of data, say an image, to certain unsupervised learning algorithms, they will then learn to group pixels together and say, gee, this sort of pixel seems to belong together, and that sort of pixel seems to belong together.

And so the images you see on the bottom — I guess you can just barely see them on there — so the images you see on the bottom are groupings — are what the algorithm has done to group certain pixels together. On a small display, it might be easier to just look at the image on the right. The two images on the bottom are two sort of identical visualizations of the same grouping of the pixels into [inaudible] regions.

And so it turns out that this sort of clustering algorithm or this sort of unsupervised learning algorithm, which learns to group pixels together, it turns out to be useful for many applications in vision, in computer vision image processing.

I'll just show you one example, and this is a rather cool one that two students, Ashutosh Saxena and Min Sun here did, which is given an image like this, right? This is actually a picture taken of the Stanford campus. You can apply that sort of clustering algorithm and group the picture into regions. Let me actually blow that up so that you can see it more clearly. Okay. So in the middle, you see the lines sort of grouping the image together, grouping the image into [inaudible] regions.

And what Ashutosh and Min did was they then applied the learning algorithm to say can we take this clustering and use it to build a 3D model of the world? And so using the clustering, they then had a learning algorithm try to learn what the 3D structure of the world looks like so that they could come up with a 3D model that you can sort of fly through, okay? Although many people used to think it's not possible to take a single image and build a 3D model, but using a learning algorithm and that sort of clustering algorithm is the first step. They were able to.

I'll just show you one more example. I like this because it's a picture of Stanford with our beautiful Stanford campus. So again, taking the same sort of clustering algorithms, taking the same sort of unsupervised learning algorithm, you can group the pixels into different regions. And using that as a pre-processing step, they eventually built this sort of 3D model of Stanford campus in a single picture. You can sort of walk into the ceiling, look around the campus. Okay? This actually turned out to be a mix of supervised and unsupervised learning, but the unsupervised learning, this sort of clustering was the first step.

So it turns out these sorts of unsupervised — clustering algorithms are actually routinely used for many different problems, things like organizing computing clusters, social network analysis, market segmentation, so if you're a marketer and you want to divide your market into different segments or different groups of people to market to them separately; even for astronomical data analysis and understanding how galaxies

are formed. These are just a sort of small sample of the applications of unsupervised learning algorithms and clustering algorithms that we'll talk about later in this class.

Just one particularly cool example of an unsupervised learning algorithm that I want to tell you about. And to motivate that, I'm gonna tell you about what's called the cocktail party problem, which is imagine that you're at some cocktail party and there are lots of people standing all over. And you know how it is, right, if you're at a large party, everyone's talking, it can be sometimes very hard to hear even the person in front of you. So imagine a large cocktail party with lots of people. So the problem is, is that all of these people talking, can you separate out the voice of just the person you're interested in talking to with all this loud background noise?

So I'll show you a specific example in a second, but here's a cocktail party that's I guess rather sparsely attended by just two people. But what we're gonna do is we'll put two microphones in the room, okay? And so because the microphones are just at slightly different distances to the two people, and the two people may speak in slightly different volumes, each microphone will pick up an overlapping combination of these two people's voices, so slightly different overlapping voices. So Speaker 1's voice may be more loud on Microphone 1, and Speaker 2's voice may be louder on Microphone 2, whatever.

But the question is, given these microphone recordings, can you separate out the original speaker's voices? So I'm gonna play some audio clips that were collected by Tai Yuan Lee at UCSD. I'm gonna actually play for you the original raw microphone recordings from this cocktail party. So this is the Microphone 1:

Microphone 1:

One, two, three, four, five, six, seven, eight, nine, ten.

Microphone 2:

Uno, dos, tres, cuatro, cinco, seis, siete, ocho, nueve, diez.

Instructor (Andrew Ng) : So it's a fascinating cocktail party with people counting from one to ten. This is the second microphone:

Microphone 1:

One, two, three, four, five, six, seven, eight, nine, ten.

Microphone 2:

Uno, dos, tres, cuatro, cinco, seis, siete, ocho, nueve, diez.

Instructor (Andrew Ng) : Okay. So in supervised learning, we don't know what the right answer is, right? So what we're going to do is take exactly the two microphone recordings you just heard and give it to an unsupervised learning algorithm and tell the algorithm which of these discover structure in the data [inaudible] or what structure is there in this data? And we actually don't know what the right answer is offhand.

So give this data to an unsupervised learning algorithm, and what the algorithm does in this case, it will discover that this data can actually be explained by two independent speakers speaking at the same time, and it can further separate out the two speakers for you. So here's Output 1 of the algorithm:

Microphone 1:

One, two, three, four, five, six, seven, eight, nine, ten.

Instructor (Andrew Ng) : And there's the second algorithm:

Microphone 2:

Uno, dos, tres, cuatro, cinco, seis, siete, ocho, nueve, diez.

Instructor (Andrew Ng): And so the algorithm discovers that, gee, the structure underlying the data is really that there are two sources of sound, and here they are. I'll show you one more example. This is a, well, this is a second sort of different pair of microphone recordings:

Microphone 1:

One, two, three, four, five, six, seven, eight, nine, ten.

Microphone 2:

[Music playing.]

Instructor (Andrew Ng): So the poor guy is not at a cocktail party. He's talking to his radio. There's the second recording:

Microphone 1:

One, two, three, four, five, six, seven, eight, nine, ten.
Microphone 2:
[Music playing.]
Instructor (Andrew Ng) : Right. And we get this data. It's the same unsupervised learning algorithm. The algorithm is actually called independent component analysis, and later in this quarter, you'll see why. And then output's the following:
Microphone 1:
One, two, three, four, five, six, seven, eight, nine, ten.
Instructor (Andrew Ng): And that's the second one:
Microphone 2:
[Music playing.]
Instructor (Andrew Ng): Okay. So it turns out that beyond solving the cocktail party algorithm, this specific class of unsupervised learning algorithms are also applied to a bunch of other problems, like in text processing or understanding functional grading and machine data, like the magneto-encephalogram would be an EEG data. We'll talk about that more when we go and describe ICA or independent component analysis algorithms, which is what you just saw.

And as an aside, this algorithm I just showed you, it seems like it must be a pretty complicated algorithm, right, to take this overlapping audio streams and separate them out. It sounds like a pretty complicated thing to do. So you're gonna ask how complicated is it really to implement an algorithm like this? It turns out if you do it in MATLAB, you can do it in one line of code.

So I got this from Samuel Wyse at Toronto, U of Toronto, and the example I showed you actually used a more complicated ICA algorithm than this. But nonetheless, I guess this is why for this class I'm going to ask you to do most of your programming in MATLAB and Octave because if you try to implement the same algorithm in C or Java or something, I can tell you from personal, painful experience, you end up writing pages and pages of code rather than relatively few lines of code. I'll also mention that it did take researchers many, many years to come up with that one line of code, so this is not easy.

So that was unsupervised learning, and then the last of the four major topics I wanna tell you about is reinforcement learning. And this refers to problems where you don't do one-shot decision-making. So, for example, in the supervised learning cancer prediction problem, you have a patient come in, you predict that the cancer is malignant or benign. And then based on your prediction, maybe the patient lives or dies, and then that's it, right? So you make a decision and then there's a consequence. You either got it right or wrong. In reinforcement learning problems, you are usually asked to make a sequence of decisions over time.

So, for example, this is something that my students and I work on. If I give you the keys to an autonomous helicopter — we actually have this helicopter here at Stanford, — how do you write a program to make it fly, right? You notice that if you make a wrong decision on a helicopter, the consequence of crashing it may not happen until much later. And in fact, usually you need to make a whole sequence of bad decisions to crash a helicopter. But conversely, you also need to make a whole sequence of good decisions in order to fly a helicopter really well.

So I'm gonna show you some fun videos of learning algorithms flying helicopters. This is a video of our helicopter at Stanford flying using a controller that was learned using a reinforcement learning algorithm. So this was done on the Stanford football field, and we'll zoom out the camera in a second. You'll sort of see the trees planted in the sky. So maybe this is one of the most difficult aerobatic maneuvers flown on any helicopter under computer control. And this controller, which is very, very hard for a human to sit down and write out, was learned using one of these reinforcement learning algorithms.

Just a word about that: The basic idea behind a reinforcement learning algorithm is this idea of what's called a reward function. What we have to think about is imagine you're trying to train a dog. So every time your dog does something good, you say, "Good dog," and you reward the dog. Every time your dog does something bad, you go, "Bad dog," right? And hopefully, over time, your dog will learn to do the right things to get more of the positive rewards, to get more of the "Good dogs" and to get fewer of the "Bad dogs."

So the way we teach a helicopter to fly or any of these robots is sort of the same thing. Every time the

helicopter crashes, we go, "Bad helicopter," and every time it does the right thing, we go, "Good helicopter," and over time it learns how to control itself so as to get more of these positive rewards.

So reinforcement learning is — I think of it as a way for you to specify what you want done, so you have to specify what is a "good dog" and what is a "bad dog" behavior. And then it's up to the learning algorithm to figure out how to maximize the "good dog" reward signals and minimize the "bad dog" punishments.

So it turns out reinforcement learning is applied to other problems in robotics. It's applied to things in web crawling and so on. But it's just cool to show videos, so let me just show a bunch of them. This learning algorithm was actually implemented by our head TA, Zico, of programming a four-legged dog. I guess Sam Shriver in this class also worked on the project and Peter Renfrew and Mike and a few others. But I guess this really is a good dog/bad dog since it's a robot dog.

The second video on the right, some of the students, I guess Peter, Zico, Tonca working on a robotic snake, again using learning algorithms to teach a snake robot to climb over obstacles.

Below that, this is kind of a fun example. Ashutosh Saxena and Jeff Michaels used learning algorithms to teach a car how to drive at reasonably high speeds off roads avoiding obstacles.

And on the lower right, that's a robot programmed by PhD student Eva Roshen to teach a sort of somewhat strangely configured robot how to get on top of an obstacle, how to get over an obstacle. Sorry. I know the video's kind of small. I hope you can sort of see it. Okay?

So I think all of these are robots that I think are very difficult to hand-code a controller for by learning these sorts of learning algorithms. You can in relatively short order get a robot to do often pretty amazing things.

Okay. So that was most of what I wanted to say today. Just a couple more last things, but let me just check what questions you have right now. So if there are no questions, I'll just close with two reminders, which are after class today or as you start to talk with other people in this class, I just encourage you again to start to form project partners, to try to find project partners to do your project with. And also, this is a good time to start forming study groups, so either talk to your friends or post in the newsgroup, but we just encourage you to try to start to do both of those today, okay? Form study groups, and try to find two other project partners.

So thank you. I'm looking forward to teaching this class, and I'll see you in a couple of days.

1.2 Machine Learning Lecture 2[2]

Figure 1.2

Instructor (Andrew Ng): All right, good morning, welcome back. So before we jump into today's material, I just have one administrative announcement, which is graders. So I guess sometime next week, we'll hand out the first homework assignment for this class.

Is this loud enough, by the way? Can people in the back hear me? No. Can you please turn up the mic a bit louder? Is this better? Is this okay? This is okay? Great.

So sometime next week, we'll hand out the first problem sets and it'll be two weeks after that, and the way we grade homework problems in this class is by some combination of TAs and graders, where graders are usually members – students currently in the class.

So in maybe about a week or so, I'll email the class to solicit applications for those of you that might be interested in becoming graders for this class, and there's usually sort of a fun thing to do. So four times

---
[2]This content is available online at <http://cnx.org/content/m45859/1.2/>.

this quarter, the TAs, and the graders, and I will spend one evening staying up late and grading all the homework problems.

For those of you who that have never taught a class before, or sort of been a grader, it's an interesting way for you to see, you know, what the other half of the teaching experience is. So the students that grade for the first time sort of get to learn about what it is that really makes a difference between a good solution and amazing solution. And to give everyone to just how we do points assignments, or what is it that causes a solution to get full marks, or just how to write amazing solutions. Becoming a grader is usually a good way to do that.

Graders are paid positions and you also get free food, and it's usually fun for us to sort of hang out for an evening and grade all the assignments. Okay, so I will send email. So don't email me yet if you want to be a grader. I'll send email to the entire class later with the administrative details and to solicit applications. So you can email us back then, to apply, if you'd be interested in being a grader.

Okay, any questions about that? All right, okay, so let's get started with today's material. So welcome back to the second lecture. What I want to do today is talk about linear regression, gradient descent, and the normal equations. And I should also say, lecture notes have been posted online and so if some of the math I go over today, I go over rather quickly, if you want to see every equation written out and work through the details more slowly yourself, go to the course homepage and download detailed lecture notes that pretty much describe all the mathematical, technical contents I'm going to go over today.

Today, I'm also going to delve into a fair amount – some amount of linear algebra, and so if you would like to see a refresher on linear algebra, this week's discussion section will be taught by the TAs and will be a refresher on linear algebra. So if some of the linear algebra I talk about today sort of seems to be going by pretty quickly, or if you just want to see some of the things I'm claiming today with our proof, if you want to just see some of those things written out in detail, you can come to this week's discussion section.

So I just want to start by showing you a fun video. Remember at the last lecture, the initial lecture, I talked about supervised learning. And supervised learning was this machine-learning problem where I said we're going to tell the algorithm what the close right answer is for a number of examples, and then we want the algorithm to replicate more of the same.

So the example I had at the first lecture was the problem of predicting housing prices, where you may have a training set, and we tell the algorithm what the "right" housing price was for every house in the training set. And then you want the algorithm to learn the relationship between sizes of houses and the prices, and essentially produce more of the "right" answer.

So let me show you a video now. Load the big screen, please. So I'll show you a video now that was from Dean Pomerleau at some work he did at Carnegie Mellon on applied supervised learning to get a car to drive itself. This is work on a vehicle known as Alvin. It was done sort of about 15 years ago, and I think it was a very elegant example of the sorts of things you can get supervised or any algorithms to do.

On the video, you hear Dean Pomerleau's voice mention and algorithm called Neural Network. I'll say a little bit about that later, but the essential learning algorithm for this is something called gradient descent, which I will talk about later in today's lecture. Let's watch the video. [Video plays]

Instructor (Andrew Ng):So two comments, right. One is this is supervised learning because it's learning from a human driver, in which a human driver shows that we're on this segment of the road, I will steer at this angle. This segment of the road, I'll steer at this angle. And so the human provides the number of "correct" steering directions to the car, and then it's the job of the car to try to learn to produce more of these "correct" steering directions that keeps the car on the road.

On the monitor display up here, I just want to tell you a little bit about what this display means. So on the upper left where the mouse pointer is moving, this horizontal line actually shows the human steering direction, and this white bar, or this white area right here shows the steering direction chosen by the human driver, by moving the steering wheel.

The human is steering a little bit to the left here indicated by the position of this white region. This second line here where Mamos is pointing, the second line here is the output of the learning algorithm, and where the learning algorithm currently thinks is the right steering direction. And right now what you're seeing is the learning algorithm just at the very beginning of training, and so there's just no idea of where

to steer. And so its output, this little white smear over the entire range of steering directions.

And as the algorithm collects more examples and learns of a time, you see it start to more confidently choose a steering direction. So let's keep watching the video. [Video plays]

Instructor (Andrew Ng): All right, so who thought driving could be that dramatic, right? Switch back to the chalkboard, please. I should say, this work was done about 15 years ago and autonomous driving has come a long way. So many of you will have heard of the DARPA Grand Challenge, where one of my colleagues, Sebastian Thrun, the winning team's drive a car across a desert by itself.

So Alvin was, I think, absolutely amazing work for its time, but autonomous driving has obviously come a long way since then. So what you just saw was an example, again, of supervised learning, and in particular it was an example of what they call the regression problem, because the vehicle is trying to predict a continuous value variables of a continuous value steering directions, we call the regression problem.

And what I want to do today is talk about our first supervised learning algorithm, and it will also be to a regression task. So for the running example that I'm going to use throughout today's lecture, you're going to return to the example of trying to predict housing prices. So here's actually a dataset collected by TA, Dan Ramage, on housing prices in Portland, Oregon.

So here's a dataset of a number of houses of different sizes, and here are their asking prices in thousands of dollars, $200,000. And so we can take this data and plot it, square feet, best price, and so you make your other dataset like that. And the question is, given a dataset like this, or given what we call a training set like this, how do you learn to predict the relationship between the size of the house and the price of the house?

So I'm actually going to come back and modify this task a little bit more later, but let me go ahead and introduce some notation, which I'll be using, actually, throughout the rest of this course. The first piece of notation is I'm going to let the lower case alphabet M denote the number of training examples, and that just means the number of rows, or the number of examples, houses, and prices we have.

And in this particular dataset, we have, what actually happens, we have 47 training examples, although I wrote down only five. Okay, so throughout this quarter, I'm going to use the alphabet M to denote the number of training examples. I'm going to use the lower case alphabet X to denote the input variables, which I'll often also call the features. And so in this case, X would denote the size of the house they were looking at.

I'm going to use Y to denote the "output" variable, which is sometimes also called a target variable, and so one pair, x, y, is what comprises one training example. In other words, one row on the table I drew just now what would be what I call one training example, and the Ith training example, in other words the Ith row in that table, I'm going to write as XI, Y, I.

Okay, and so in this notation they're going to use this superscript I is not exponentiation. So this is not X to the power of IY to the power of I. In this notation, the superscript I in parentheses is just sort of an index into the Ith row of my list of training examples.

So in supervised learning, this is what we're going to do. We're given a training set, and we're going to feed our training set, comprising our M training example, so 47 training examples, into a learning algorithm. Okay, and our algorithm then has output function that is by tradition, and for historical reasons, is usually denoted lower case alphabet H, and is called a hypothesis. Don't worry too much about whether the term hypothesis has a deep meaning. It's more a term that's used for historical reasons.

And the hypothesis's job is to take this input. There's some new [inaudible]. What the hypothesis does is it takes this input, a new living area in square feet saying and output estimates the price of this house. So the hypothesis H maps from inputs X to outputs Y. So in order to design a learning algorithm, the first thing we have to decide is how we want to represent the hypothesis, right.

And just for this purposes of this lecture, for the purposes of our first learning algorithm, I'm going to use a linear representation for the hypothesis. So I'm going to represent my hypothesis as H of X equals theta zero, plus theta 1X, where X here is an input feature, and so that's the size of the house we're considering.

And more generally, come back to this, more generally for many regression problems we may have more than one input feature. So for example, if instead of just knowing the size of the houses, if we know also the number of bedrooms in these houses, let's say, then, so if our training set also has a second feature, the number of bedrooms in the house, then you may, let's say X1 denote the size and square feet. Let X have

script two denote the number of bedrooms, and then I would write the hypothesis, H of X, as theta rho plus theta 1X1 plus theta 2X2.

Okay, and sometimes when I went to take the hypothesis H, and when I went to make this dependent on the theta is explicit, I'll sometimes write this as H subscript theta of X. And so this is the price that my hypothesis predicts a house with features X costs. So given the new house with features X, a certain size and a certain number of bedrooms, this is going to be the price that my hypothesis predicts this house is going to cost.

One final piece of notation, so for conciseness, just to write this a bit more compactly I'm going to take the convention of defining X0 to be equal to one, and so I can now write H of X to be equal to sum from I equals one to two of theta I, oh sorry, zero to two, theta I, X I. And if you think of theta as an X, as vectors, then this is just theta transpose X.

And the very final piece of notation is I'm also going to let lower case N be the number of features in my learning problem. And so this actually becomes a sum from I equals zero to N, where in this example if you have two features, N would be equal to two.

All right, I realize that was a fair amount of notation, and as I proceed through the rest of the lecture today, or in future weeks as well, if some day you're looking at me write a symbol and you're wondering, gee, what was that simple lower case N again? Or what was that lower case X again, or whatever, please raise hand and I'll answer. This is a fair amount of notation. We'll probably all get used to it in a few days and we'll standardize notation and make a lot of our descriptions of learning algorithms a lot easier.

But again, if you see me write some symbol and you don't quite remember what it means, chances are there are others in this class who've forgotten too. So please raise your hand and ask if you're ever wondering what some symbol means. Any questions you have about any of this?

Yeah?

Student: The variable can be anything? [Inaudible]?

Instructor (Andrew Ng): Say that again.

Student:[inaudible] zero theta one?

Instructor (Andrew Ng): Right, so, well let me – this was going to be next, but the theta or the theta Is are called the parameters. The thetas are called the parameters of our learning algorithm and theta zero, theta one, theta two are just real numbers. And then it is the job of the learning algorithm to use the training set to choose or to learn appropriate parameters theta.

Okay, is there other questions?

Student: What does [inaudible]?

Instructor (Andrew Ng): Oh, transpose. Oh yeah, sorry. When [inaudible] theta and theta transpose X, theta [inaudible].

Student: Is this like a [inaudible] hypothesis [inaudible], or would you have higher orders? Or would theta [inaudible]?

Instructor (Andrew Ng): All great questions. The answer – so the question was, is this a typical hypothesis or can theta be a function of other variables and so on. And the answer is sort of yes. For now, just for this first learning algorithm we'll talk about using a linear hypothesis class. A little bit actually later this quarter, we'll talk about much more complicated hypothesis classes, and we'll actually talk about higher order functions as well, a little bit later today.

Okay, so for the learning problem then. How do we chose the parameters theta so that our hypothesis H will make accurate predictions about all the houses. All right, so one reasonable thing to do seems to be, well, we have a training set. So – and just on the training set, our hypothesis will make some prediction, predictions of the housing prices, of the prices of the houses in the training set.

So one thing we could do is just try to make the predictions of a learning algorithm accurate on a training set. So given some features, X, and some correct prices, Y, we might want to make that theta square difference between the prediction of the algorithm and the actual price [inaudible].

So to choose parameters theta, unless we want to minimize over the parameters theta, so the squared area between the predicted price and the actual price. And so going to fill this in. We have M training examples. So the sum from I equals one through M of my M training examples, of price predicted on the

Ith house in my training set. Mine is the actual target variable. Mine is actual price on the Ith training example.

And by convention, instead of minimizing this sum of the squared differences, I'm just going to put a one-half there, which will simplify some of the math we do later. Okay, and so let me go ahead and define J of theta to be equal to just the same, one-half sum from I equals one through M on the number of training examples, of the value predicted by my hypothesis minus the actual value.

And so what we'll do, let's say, is minimize as a function of the parameters of theta, this quantity J of theta. I should say, to those of you who have taken sort of linear algebra classes, or maybe basic statistics classes, some of you may have seen things like these before and seen least [inaudible] regression or [inaudible] squares.

Many of you will not have seen this before. I think some of you may have seen it before, but either way, regardless of whether you've seen it before, let's keep going. Just for those of you that have seen it before, I should say eventually, we'll actually show that this algorithm is a special case of a much broader class of algorithms. But let's keep going. We'll get there eventually.

So I'm going to talk about a couple of different algorithms for performing that minimization over theta of J of theta. The first algorithm I'm going to talk about is a search algorithm, where the basic idea is we'll start with some value of my parameter vector theta. Maybe initialize my parameter vector theta to be the vector of all zeros, and excuse me, have to correct that. I sort of write zero with an arrow on top to denote the vector of all zeros.

And then I'm going to keep changing my parameter vector theta to reduce J of theta a little bit, until we hopefully end up at the minimum with respect to theta of J of theta. So switch the laptops please, and lower the big screen. So let me go ahead and show you an animation of this first algorithm for minimizing J of theta, which is an algorithm called grading and descent.

So here's the idea. You see on the display a plot and the axes, the horizontal axes are theta zero and theta one. That's usually – minimize J of theta, which is represented by the height of this plot. So the surface represents the function J of theta and the axes of this function, or the inputs of this function are the parameters theta zero and theta one, written down here below.

So here's the gradient descent algorithm. I'm going to choose some initial point. It could be vector of all zeros or some randomly chosen point. Let's say we start from that point denoted by the star, by the cross, and now I want you to imagine that this display actually shows a 3D landscape. Imagine you're all in a hilly park or something, and this is the 3D shape of, like, a hill in some park.

So imagine you're actually standing physically at the position of that star, of that cross, and imagine you can stand on that hill, right, and look all 360 degrees around you and ask, if I were to take a small step, what would allow me to go downhill the most? Okay, just imagine that this is physically a hill and you're standing there, and would look around ask, "If I take a small step, what is the direction of steepest descent, that would take me downhill as quickly as possible?"

So the gradient descent algorithm does exactly that. I'm going to take a small step in this direction of steepest descent, or the direction that the gradient turns out to be. And then you take a small step and you end up at a new point shown there, and it would keep going. You're now at a new point on this hill, and again you're going to look around you, look all 360 degrees around you, and ask, "What is the direction that would take me downhill as quickly as possible?"

And we want to go downhill as quickly as possible, because we want to find the minimum of J of theta. So you do that again. You can take another step, okay, and you sort of keep going until you end up at a local minimum of this function, J of theta. One property of gradient descent is that where you end up – in this case, we ended up at this point on the lower left hand corner of this plot.

But let's try running gradient descent again from a different position. So that was where I started gradient descent just now. Let's rerun gradient descent, but using a slightly different initial starting point, so a point slightly further up and further to the right. So it turns out if you run gradient descent from that point, then if you take a steepest descent direction again, that's your first step.

And if you keep going, it turns out that with a slightly different initial starting point, you can actually end up at a completely different local optimum. Okay, so this is a property of gradient descent, and we'll

come back to it in a second. So be aware that gradient descent can sometimes depend on where you initialize your parameters, theta zero and theta one.

Switch back to the chalkboard, please. Let's go ahead and work out the math of the gradient descent algorithm. Then we'll come back and revisit this issue of local optimum. So here's the gradient descent algorithm.

We're going to take a repeatedly take a step in the direction of steepest descent, and it turns out that you can write that as [inaudible], which is we're going to update the parameters theta as theta I minus the partial derivative with respect to theta I, J of Theta. Okay, so this is how we're going to update the I parameter, theta I, how we're going to update Theta I on each iteration of gradient descent.

Just a point of notation, I use this colon equals notation to denote setting a variable on the left hand side equal to the variable on the right hand side. All right, so if I write A colon equals B, then what I'm saying is, this is part of a computer program, or this is part of an algorithm where we take the value of B, the value on the right hand side, and use that to overwrite the value on the left hand side.

In contrast, if I write A equals B, then this is an assertion of truth. I'm claiming that the value of A is equal to the value of B, whereas this is computer operation where we overwrite the value of A. If I write A equals B then I'm asserting that the values of A and B are equal.

So let's see, this algorithm sort of makes sense – well, actually let's just move on. Let's just go ahead and take this algorithm and apply it to our problem. And to work out gradient descent, let's take gradient descent and just apply it to our problem, and this being the first somewhat mathematical lecture, I'm going to step through derivations much more slowly and carefully than I will later in this quarter. We'll work through the steps of these in much more detail than I will later in this quarter.

Let's actually work out what this gradient descent rule is. So – and I'll do this just for the case of, if we had only one training example. Okay, so in this case we need to work out what the partial derivative with respect to the parameter theta I of J of theta. If we have only one training example then J of theta is going to be one-half of script theta, of X minus Y, script. So if you have only one training example comprising one pair, X, Y, then this is what J of theta is going to be.

And so taking derivatives, you have one-half something squared. So the two comes down. So you have two times one-half times theta of X minus Y, and then by the [inaudible] derivatives, we also must apply this by the derivative of what's inside the square. Right, the two and the one-half cancel. So this leaves [inaudible] times that, theta zero, X zero plus [inaudible].

Okay, and if you look inside this sum, we're taking the partial derivative of this sum with respect to the parameter theta I. But all the terms in the sum, except for one, do not depend on theta I. In this sum, the only term that depends on theta I will be some term here of theta I, X I. And so we take the partial derivative with respect to theta I, X I – take the partial derivative with respect to theta I of this term theta I, X I, and so you get that times X I.

Okay, and so this gives us our learning rule, right, of theta I gets updated as theta I minus alpha times that. Okay, and this Greek alphabet alpha here is a parameter of the algorithm called the learning rate, and this parameter alpha controls how large a step you take. So you're standing on the hill. You decided what direction to take a step in, and so this parameter alpha controls how aggressive – how large a step you take in this direction of steepest descent.

And so if you – and this is a parameter of the algorithm that's often set by hand. If you choose alpha to be too small than your steepest descent algorithm will take very tiny steps and take a long time to converge. If alpha is too large then the steepest descent may actually end up overshooting the minimum, if you're taking too aggressive a step.

Yeah?

Student:[Inaudible].

Instructor (Andrew Ng):Say that again?

Student: Isn't there a one over two missing somewhere?

Instructor (Andrew Ng):Is there a one-half missing?

Student:I was [inaudible].

Instructor (Andrew Ng):Thanks. I do make lots of errors in that. Any questions about this?

All right, so let me just wrap this property into an algorithm. So over there I derived the algorithm where you have just one training example, more generally for M training examples, gradient descent becomes the following. We're going to repeat until convergence the following step.

Okay, theta I gets updated as theta I and I'm just writing out the appropriate equation for M examples rather than one example. Theta I gets updated. Theta I minus alpha times the sum from I equals one to M. Okay, and I won't bother to show it, but you can go home and sort of verify for yourself that this summation here, this is indeed the partial derivative with respect to theta I of J of theta, where if you use the original definition of J of theta for when you have M training examples.

Okay, so I'm just going to show – switch back to the laptop display. I'm going to show you what this looks like when you run the algorithm. So it turns out that for the specific problem of linear regression, or ordinary release squares, which is what we're doing today, the function J of theta actually does not look like this nasty one that I'll show you just now with a multiple local optima.

In particular, it turns out for ordinary release squares, the function J of theta is – it's just a quadratic function. And so we'll always have a nice bow shape, like what you see up here, and only have one global minimum with no other local optima.

So when you run gradient descent, here are actually the contours of the function J. So the contours of a bow shaped function like that are going to be ellipses, and if you run gradient descent on this algorithm, here's what you might get. Let's see, so I initialize the parameters. So let's say randomly at the position of that cross over there, right, that cross on the upper right.

And so after one iteration of gradient descent, as you change the space of parameters, so if that's the result of one step of gradient descent, two steps, three steps, four steps, five steps, and so on, and it, you know, converges easily, rapidly to the global minimum of this function J of theta.

Okay, and this is a property of [inaudible] regression with a linear hypothesis cost. The function, J of theta has no local optima. Yes, question?

Student: Is the alpha changing every time? Because the step is not [inaudible].

Instructor (Andrew Ng): So it turns out that – yes, so it turns out – this was done with a – this is with a fake value of alpha, and one of the properties of gradient descent is that as you approach the local minimum, it actually takes smaller and smaller steps so they'll converge. And the reason is, the update is – you update theta by subtracting from alpha times the gradient. And so as you approach the local minimum, the gradient also goes to zero.

As you approach the local minimum, at the local minimum the gradient is zero, and as you approach the local minimum, the gradient also gets smaller and smaller. And so gradient descent will automatically take smaller and smaller steps as you approach the local minimum. Make sense?

And here's the same plot – here's actually a plot of the housing prices data. So here, lets you initialize the parameters to the vector of all zeros, and so this blue line at the bottom shows the hypothesis with the parameters of initialization. So initially theta zero and theta one are both zero, and so your hypothesis predicts that all prices are equal to zero.

After one iteration of gradient descent, that's the blue line you get. After two iterations, three, four, five, and after a few more iterations, excuse me, it converges, and you've now found the least square fit for the data. Okay, let's switch back to the chalkboard. Are there questions about this? Yeah?

Student:[Inaudible] iteration, do we mean that we run each sample – all the sample cases [inaudible] the new values?

Instructor (Andrew Ng): Yes, right.

Student: And converged means that the value will be the same [inaudible] roughly the same?

Instructor (Andrew Ng): Yeah, so this is sort of a question of how do you test the convergence. And there's different ways of testing for convergence. One is you can look at two different iterations and see if theta has changed a lot, and if theta hasn't changed much within two iterations, you may say it's sort of more or less converged.

Something that's done maybe slightly more often is look at the value of J of theta, and if J of theta – if the quantity you're trying to minimize is not changing much anymore, then you might be inclined to believe it's converged. So these are sort of standard heuristics, or standard rules of thumb that are often used to

decide if gradient descent has converged.

Yeah?

Student:I may have missed something, but especially in [inaudible] descent. So one feature [inaudible] curve and can either go this way or that way. But the math at incline [inaudible] where that comes in. When do you choose whether you go left, whether you're going this way or that way?

Instructor (Andrew Ng):I see. It just turns out that – so the question is, how is gradient descent looking 360 around you and choosing the direction of steepest descent. So it actually turns out – I'm not sure I'll answer the second part, but it turns out that if you stand on the hill and if you – it turns out that when you compute the gradient of the function, when you compute the derivative of the function, then it just turns out that that is indeed the direction of steepest descent.

By the way, I just want to point out, you would never want to go in the opposite direction because the opposite direction would actually be the direction of steepest ascent, right. So as it turns out – maybe the TAs can talk a bit more about this at the section if there's interest. It turns out, when you take the derivative of a function, the derivative of a function sort of turns out to just give you the direction of steepest descent.

And so you don't explicitly look all 360 degrees around you. You sort of just compute the derivative and that turns out to be the direction of steepest descent. Yeah, maybe the TAs can talk a bit more about this on Friday.

Okay, let's see, so let me go ahead and give this algorithm a specific name. So this algorithm here is actually called batch gradient descent, and the term batch isn't a great term, but the term batch refers to the fact that on every step of gradient descent you're going to look at your entire training set. You're going to perform a sum over your M training examples.

So [inaudible] descent often works very well. I use it very often, and it turns out that sometimes if you have a really, really large training set, imagine that instead of having 47 houses from Portland, Oregon in our training set, if you had, say, the U.S. Census Database or something, with U.S. census size databases you often have hundreds of thousands or millions of training examples.

So if M is a few million then if you're running batch rate and descent, this means that to perform every step of gradient descent you need to perform a sum from J equals one to a million. That's sort of a lot of training examples where your computer programs have to look at, before you can even take one step downhill on the function J of theta.

So it turns out that when you have very large training sets, you should write down an alternative algorithm that is called [inaudible] gradient descent. Sometimes I'll also call it incremental gradient descent, but the algorithm is as follows. Again, it will repeat until convergence and will iterate for J equals one to M, and will perform one of these sort of gradient descent updates using just the J training example.

Oh, and as usual, this is really – you update all the parameters data runs. You perform this update for all values of I. For I indexes and the parameter vectors, you just perform this update, all of your parameters simultaneously. And the advantage of this algorithm is that in order to start learning, in order to start modifying the parameters, you only need to look at your first training examples.

You should look at your first training example and perform an update using the derivative of the error with respect to just your first training example, and then you look at your second training example and perform another update. And you sort of keep adapting your parameters much more quickly without needing to scan over your entire U.S. Census database before you can even start adapting parameters.

So let's see, for launch data sets, so constantly gradient descent is often much faster, and what happens is that constant gradient descent is that it won't actually converge to the global minimum exactly, but if these are the contours of your function, then after you run the constant gradient descent, you sort of tend to wander around.

And you may actually end up going uphill occasionally, but your parameters will sort of tender to wander to the region closest to the global minimum, but sort of keep wandering around a little bit near the region of the global [inaudible]. And often that's just fine to have a parameter that wanders around a little bit the global minimum. And in practice, this often works much faster than back gradient descent, especially if you have a large training set.

Okay, I'm going to clean a couple of boards. While I do that, why don't you take a look at the equations,

and after I'm done cleaning the boards, I'll ask what questions you have.

Okay, so what questions do you have about all of this?

Student:[Inaudible] is it true – are you just sort of rearranging the order that you do the computation? So do you just use the first training example and update all of the theta Is and then step, and then update with the second training example, and update all the theta Is, and then step? And is that why you get sort of this really – ?

Instructor (Andrew Ng):Let's see, right. So I'm going to look at my first training example and then I'm going to take a step, and then I'm going to perform the second gradient descent updates using my new parameter vector that has already been modified using my first training example. And then I keep going.

Make sense? Yeah?

Student: So in each update of all the theta Is, you're only using –

Instructor (Andrew Ng):One training example.

Student: One training example.

Student:[Inaudible]?

Instructor (Andrew Ng):Let's see, it's definitely a [inaudible]. I believe this theory that sort of supports that as well. Yeah, the theory that supports that, the [inaudible] of theorem is, I don't remember.

Okay, cool. So in what I've done so far, I've talked about an iterative algorithm for performing this minimization in terms of J of theta. And it turns out that there's another way for this specific problem of least squares regression, of ordinary least squares. It turns out there's another way to perform this minimization of J of theta that allows you to solve for the parameters theta in close form, without needing to run an iterative algorithm.

And I know some of you may have seen some of what I'm about to do before, in like an undergraduate linear algebra course, and the way it's typically done requires [inaudible] projections, or taking lots of derivatives and writing lots of algebra. What I'd like to do is show you a way to derive the closed form solution of theta in just a few lines of algebra.

But to do that, I'll need to introduce a new notation for matrix derivatives, and it turns out that, sort of, the notation I'm about to define here just in my own personal work has turned out to be one of the most useful things that I actually use all the time, to have a notation of how to take derivatives with respect to matrixes, so that you can solve for the minimum of J of theta with, like, a few lines of algebra rather than writing out pages and pages of matrices and derivatives.

So then we're going to define this new notation first and then we'll go ahead and work out the minimization. Given a function J, since J is a function of a vector of parameters theta, right, I'm going to define the derivative of the gradient of J with respect to theta, as self of vector. Okay, and so this is going to be an N plus one dimensional vector. Theta is an n plus one dimensional vector with indices ranging from zero to N. And so I'm going to define this derivative to be equal to that.

Okay, and so we can actually rewrite the gradient descent algorithm as follows. This is batch gradient descent, and we write gradient descent as updating the parameter vector theta – notice there's no subscript I now – updating the parameter vector theta as the previous parameter minus alpha times the gradient.

Okay, and so in this equation all of these quantities, theta, and this gradient vector, all of these are n plus one dimensional vectors. I was using the boards out of order, wasn't I? So more generally, if you have a function F that maps from the space of matrices A, that maps from, say, the space of N by N matrices to the space of real numbers. So if you have a function, F of A, where A is an N by N matrix.

So this function is matched from matrices to real numbers, the function that takes this input to matrix. Let me define the derivative with respect to F of the matrix A. Now, I'm just taking the gradient of F with respect to its input, which is the matrix. I'm going to define this itself to be a matrix.

Okay, so the derivative of F with respect to A is itself a matrix, and the matrix contains all the partial derivatives of F with respect to the elements of A. One more definition is if A is a square matrix, so if A is an n by n matrix, number of rows equals number of columns, let me define the trace of A to be equal to the sum of A's diagonal elements. So this is just sum over I of A, I, I.

For those of you that haven't seen this sort of operator notation before, you can think of trace of A as the trace operator applied to the square matrix A, but it's more commonly written without the parentheses.

So I usually write trace of A like this, and this just means the sum of diagonal elements.

So here are some facts about the trace operator and about derivatives, and I'm just going to write these without proof. You can also have the TAs prove some of them in the discussion section, or you can actually go home and verify the proofs of all of these.

It turns out that given two matrices, A and B, the trace of the matrix A times B is equal to the trace of B, A. Okay, I'm not going to prove this, but you should be able to go home and prove this yourself without too much difficulty. And similarly, the trace of a product of three matrices, so if you can take the matrix at the end and cyclically permeate it to the front.

So trace of A times B, times C, is equal to the trace of C, A, B. So take the matrix C at the back and move it to the front, and this is also equal to the trace of B, C. Take the matrix B and move it to the front.

Okay, also, suppose you have a function F of A which is defined as a trace of A, B. Okay, so this is, right, the trace is a real number. So the trace of A, B is a function that takes this input of matrix A and output a real number. So then the derivative with respect to the matrix A of this function of trace A, B, is going to be B transposed. And this is just another fact that you can prove by yourself by going back and referring to the definitions of traces and matrix derivatives. I'm not going to prove it. You should work it out.

Okay, and lastly a couple of easy ones. The trace of A is equal to the trace of A transposed because the trace is just the sum of diagonal elements. And so if you transpose the matrix, the diagonal, then there's no change. And if lower case A is a real number, then the trace of a real number is just itself. So think of a real number as a one by one matrix. So the trace of a one by one matrix is just whatever that real number is.

And lastly, this is a somewhat tricky one. The derivative with respect to the matrix A of the trace of A, B, A, transpose C is equal to C, A, B plus C transposed A, B transposed. And I won't prove that either. This is sort of just algebra. Work it out yourself.

Okay, and so the key facts I'm going to use again about traces and matrix derivatives, I'll use five. Ten minutes. Okay, so armed with these things I'm going to figure out – let's try to come up with a quick derivation for how to minimize J of theta as a function of theta in closed form, and without needing to use an iterative algorithm.

So work this out. Let me define the matrix X. This is called the design matrix. To be a matrix containing all the inputs from my training set. So X 1 was the vector of inputs to the vector of features for my first training example. So I'm going to set X 1 to be the first row of this matrix X, set my second training example is in place to be the second variable, and so on.

And I have M training examples, and so that's going to be my design matrix X. Okay, this is defined as matrix capital X as follows, and so now, let me take this matrix X and multiply it by my parameter vector theta. This derivation will just take two or three sets. So X times theta – remember how matrix vector multiplication goes. You take this vector and you multiply it by each of the rows of the matrix.

So X times theta is just going to be X 1 transposed theta, dot, dot, dot, down to X M, transposed theta. And this is, of course, just the predictions of your hypothesis on each of your M training examples. Then we also defined the Y vector to be the vector of all the target values Y1 through YM in my training set. Okay, so Y vector is an M dimensional vector.

So X theta minus Y contained the math from the previous board, this is going to be, right, and now, X theta minus Y, this is a vector. This is an M dimensional vector in M training examples, and so I'm actually going to take this vector and take this inner product with itself.

Okay, so we call that if Z is a vector than Z transpose Z is just sum over I, ZI squared. Right, that's how you take the inner product of a vector with a sum. So you want to take this vector, X theta minus Y, and take the inner product of this vector with itself, and so that gives me sum from I equals one to M, H, F, X, I, minus Y squared. Okay, since it's just the sum of the squares of the elements of this vector.

And put a half there for the emphasis. This is our previous definition of J of theta. Okay, yeah?

Student:[Inaudible]?

Instructor (Andrew Ng): Yeah, I threw a lot of notations at you today. So M is the number of training examples and the number of training examples runs from one through M, and then is the feature vector that runs from zero through N. Does that make sense?

So this is the sum from one through M. It's sort of theta transpose X that's equal to sum from J equals

zero through N of theta J, X, J. Does that make sense? It's the feature vectors that index from zero through N where X, zero is equal to one, whereas the training example is actually indexed from one through M.

So let me clean a few more boards and take another look at this, make sure it all makes sense. Okay, yeah?

Student:[Inaudible] the Y inside the parentheses, shouldn't that be [inaudible]?

Instructor (Andrew Ng): Oh, yes, thank you. Oh is that what you meant? Yes, thank you. Great, I training example. Anything else? Cool. So we're actually nearly done with this derivation. We would like to minimize J of theta with respect to theta and we've written J of theta fairly compactly using this matrix vector notation.

So in order to minimize J of theta with respect to theta, what we're going to do is take the derivative with respect to theta of J of theta, and set this to zero, and solve for theta. Okay, so we have derivative with respect to theta of that is equal to – I should mention there will be some steps here that I'm just going to do fairly quickly without proof.

So is it really true that the derivative of half of that is half of the derivative, and I already exchanged the derivative and the one-half. In terms of the answers, yes, but later on you should go home and look through the lecture notes and make sure you understand and believe why every step is correct. I'm going to do things relatively quickly here and you can work through every step yourself more slowly by referring to the lecture notes.

Okay, so that's equal to – I'm going to expand now this quadratic function. So this is going to be, okay, and this is just sort of taking a quadratic function and expanding it out by multiplying the [inaudible]. And again, work through the steps later yourself if you're not quite sure how I did that.

So this thing, this vector, vector product, right, this quantity here, this is just J of theta and so it's just a real number, and the trace of a real number is just itself.

Student:[Inaudible].

Instructor (Andrew Ng): Oh, thanks, Dan. Cool, great. So this quantity in parentheses, this is J of theta and it's just a real number. And so the trace of a real number is just the same real number. And so you can sort of take a trace operator without changing anything. And this is equal to one-half derivative with respect to theta of the trace of – by the second permutation property of trace. You can take this theta at the end and move it to the front.

So this is going to be trace of theta times theta transposed, X transpose X minus derivative with respect to theta of the trace of – I'm going to take that and bring it to the – oh, sorry. Actually, this thing here is also a real number and the transpose of a real number is just itself. Right, so take the transpose of a real number without changing anything.

So let me go ahead and just take the transpose of this. A real number transposed itself is just the same real number. So this is minus the trace of, taking the transpose of that. Here's Y transpose X theta, then minus [inaudible] theta. Okay, and this last quantity, Y transpose Y. It doesn't actually depend on theta. So when I take the derivative of this last term with respect to theta, it's just zero. So just drop that term.

And lastly, well, the derivative with respect to theta of the trace of theta, theta transposed, X transpose X. I'm going to use one of the facts I wrote down earlier without proof, and I'm going to let this be A. There's an identity matrix there, so this is A, B, A transpose C, and using a rule that I've written down previously that you'll find in lecture notes, because it's still on one of the boards that you had previously, this is just equal to X transpose X theta.

So this is C, A, B, which is sort of just the identity matrix, which you can ignore, plus X transpose X theta where this is now C transpose C, again times the identity which we're going to ignore, times B transposed. And the matrix X transpose X is the metric, so C transpose is equal to C.

Similarly, the derivative with respect to theta of the trace of Y transpose theta X, this is the derivative with respect to matrix A of the trace of B, A and this is just X transpose Y. This is just B transposed, again, by one of the rules that I wrote down earlier. And so if you plug this back in, we find, therefore, that the derivative – wow, this board's really bad.

So if you plug this back into our formula for the derivative of J, you find that the derivative with respect to theta of J of theta is equal to one-half X transpose theta, plus X transpose X theta, minus X transpose

Y, minus X transpose Y, which is just X transpose X theta minus X [inaudible].

Okay, so we set this to zero and we get that, which is called a normal equation, and we can now solve this equation for theta in closed form. That's X transpose X theta, inverse times X transpose Y. And so this gives us a way for solving for the least square fit to the parameters in closed form, without needing to use an [inaudible] descent.

Okay, and using this matrix vector notation, I think, I don't know, I think we did this whole thing in about ten minutes, which we couldn't have if I was writing out reams of algebra. Okay, some of you look a little bit dazed, but this is our first learning hour. Aren't you excited? Any quick questions about this before we close for today?

Student:[Inaudible].

Instructor (Andrew Ng): Say that again.

Student: What you derived, wasn't that just [inaudible] of X?

Instructor (Andrew Ng): What inverse?

Student: Pseudo inverse.

Instructor (Andrew Ng): Pseudo inverse?

Student: Pseudo inverse.

Instructor (Andrew Ng): Yeah, I turns out that in cases, if X transpose X is not invertible, than you use the pseudo inverse minimized to solve this. But it turns out X transpose X is not invertible. That usually means your features were dependent. It usually means you did something like repeat the same feature twice in your training set. So if this is not invertible, it turns out the minimum is obtained by the pseudo inverses of the inverse.

If you don't know what I just said, don't worry about it. It usually won't be a problem. Anything else?

Student: On the second board [inaudible]?

Instructor (Andrew Ng): Let me take that off. We're running over. Let's close for today and if they're other questions, I'll take them after.

[End of Audio]

Duration: 79 minutes

1.3 Machine Learning Lecture 3[3]

Figure 1.3

Instructor (Andrew Ng): Okay. Good morning and welcome back to the third lecture of this class. So here's what I want to do today, and some of the topics I do today may seem a little bit like I'm jumping, sort of, from topic to topic, but here's, sort of, the outline for today and the illogical flow of ideas. In the last lecture, we talked about linear regression and today I want to talk about sort of an adaptation of that called locally weighted regression. It's very a popular algorithm that's actually one of my former mentors probably favorite machine learning algorithm.

We'll then talk about a probable second interpretation of linear regression and use that to move onto our first classification algorithm, which is logistic regression; take a brief digression to tell you about something called the perceptron algorithm, which is something we'll come back to, again, later this quarter; and time allowing I hope to get to Newton's method, which is an algorithm for fitting logistic regression models.

So this is recap where we're talking about in the previous lecture, remember the notation I defined was that I used this X superscript I, Y superscript I to denote the I training example. And when we're talking

---
[3]This content is available online at <http://cnx.org/content/m45874/1.3/>.

about linear regression or linear least squares, we use this to denote the predicted value of "by my hypothesis H" on the input XI. And my hypothesis was franchised by the vector of grams as theta and so we said that this was equal to some from theta J, si J, and more theta transpose X. And we had the convention that X subscript Z is equal to one so this accounts for the intercept term in our linear regression model. And lowercase n here was the notation I was using for the number of features in my training set. Okay? So in the example when trying to predict housing prices, we had two features, the size of the house and the number of bedrooms. We had two features and there was – little n was equal to two. So just to finish recapping the previous lecture, we defined this quadratic cos function J of theta equals one-half, something I equals one to m, theta of XI minus YI squared where this is the sum over our m training examples and my training set. So lowercase m was the notation I've been using to denote the number of training examples I have and the size of my training set. And at the end of the last lecture, we derive the value of theta that minimizes this enclosed form, which was X transpose X inverse X transpose Y. Okay?

So as we move on in today's lecture, I'll continue to use this notation and, again, I realize this is a fair amount of notation to all remember, so if partway through this lecture you forgot – if you're having trouble remembering what lowercase m is or what lowercase n is or something please raise your hand and ask. When we talked about linear regression last time we used two features. One of the features was the size of the houses in square feet, so the living area of the house, and the other feature was the number of bedrooms in the house. In general, we apply a machine-learning algorithm to some problem that you care about. The choice of the features will very much be up to you, right? And the way you choose your features to give the learning algorithm will often have a large impact on how it actually does. So just for example, the choice we made last time was X1 equal this size, and let's leave this idea of the feature of the number of bedrooms for now, let's say we don't have data that tells us how many bedrooms are in these houses. One thing you could do is actually define – oh, let's draw this out. And so, right? So say that was the size of the house and that's the price of the house. So if you use this as a feature maybe you get theta zero plus theta 1, X1, this, sort of, linear model. If you choose – let me just copy the same data set over, right? You can define the set of features where X1 is equal to the size of the house and X2 is the square of the size of the house. Okay? So X1 is the size of the house in say square footage and X2 is just take whatever the square footage of the house is and just square that number, and this would be another way to come up with a feature, and if you do that then the same algorithm will end up fitting a quadratic function for you. Theta 2, XM squared. Okay? Because this is actually X2. And depending on what the data looks like, maybe this is a slightly better fit to the data.

You can actually take this even further, right? Which is – let's see. I have seven training examples here, so you can actually maybe fit up to six for the polynomial. You can actually fill a model theta zero plus theta one, X1 plus theta two, X squared plus up to theta six. X to the power of six and theta six are the polynomial to these seven data points. And if you do that you find that you come up with a model that fits your data exactly. This is where, I guess, in this example I drew, we have seven data points, so if you fit a six model polynomial you can, sort of, fit a line that passes through these seven points perfectly. And you probably find that the curve you get will look something like that. And on the one hand, this is a great model in a sense that it fits your training data perfectly. On the other hand, this is probably not a very good model in the sense that none of us seriously think that this is a very good predictor of housing prices as a function of the size of the house, right?

So we'll actually come back to this later. It turns out of the models we have here; I feel like maybe the quadratic model fits the data best. Whereas the linear model looks like there's actually a bit of a quadratic component in this data that the linear function is not capturing. So we'll actually come back to this a little bit later and talk about the problems associated with fitting models that are either too simple, use two small a set of features, or on the models that are too complex and maybe use too large a set of features. Just to give these a name, we call this the problem of underfitting and, very informally, this refers to a setting where there are obvious patterns that – where there are patterns in the data that the algorithm is just failing to fit. And this problem here we refer to as overfitting and, again, very informally, this is when the algorithm is fitting the idiosyncrasies of this specific data set, right? It just so happens that of the seven houses we sampled in Portland, or wherever you collect data from, that house happens to be a bit more expensive,

that house happened on the less expensive, and by fitting six for the polynomial we're, sort of, fitting the idiosyncratic properties of this data set, rather than the true underlying trends of how housing prices vary as the function of the size of house. Okay?

So these are two very different problems. We'll define them more formally me later and talk about how to address each of these problems, but for now I hope you appreciate that there is this issue of selecting features. So if you want to just teach us the learning problems there are a few ways to do so. We'll talk about feature selection algorithms later this quarter as well. So automatic algorithms for choosing what features you use in a regression problem like this. What I want to do today is talk about a class of algorithms called non-parametric learning algorithms that will help to alleviate the need somewhat for you to choose features very carefully. Okay? And this leads us into our discussion of locally weighted regression. And just to define the term, linear regression, as we've defined it so far, is an example of a parametric learning algorithm. Parametric learning algorithm is one that's defined as an algorithm that has a fixed number of parameters that fit to the data. Okay? So in linear regression we have a fix set of parameters theta, right? That must fit to the data. In contrast, what I'm gonna talk about now is our first non-parametric learning algorithm. The formal definition, which is not very intuitive, so I've replaced it with a second, say, more intuitive. The, sort of, formal definition of the non-parametric learning algorithm is that it's an algorithm where the number of parameters goes with M, with the size of the training set. And usually it's defined as a number of parameters grows linearly with the size of the training set. This is the formal definition. A slightly less formal definition is that the amount of stuff that your learning algorithm needs to keep around will grow linearly with the training sets or, in another way of saying it, is that this is an algorithm that we'll need to keep around an entire training set, even after learning. Okay? So don't worry too much about this definition. But what I want to do now is describe a specific non-parametric learning algorithm called locally weighted regression. Which also goes by a couple of other names – which also goes by the name of Loess for self-hysterical reasons. Loess is usually spelled L-O-E-S-S, sometimes spelled like that, too. I just call it locally weighted regression. So here's the idea. This will be an algorithm that allows us to worry a little bit less about having to choose features very carefully. So for my motivating example, let's say that I have a training site that looks like this, okay? So this is X and that's Y. If you run linear regression on this and you fit maybe a linear function to this and you end up with a more or less flat, straight line, which is not a very good fit to this data. You can sit around and stare at this and try to decide whether the features are used right. So maybe you want to toss in a quadratic function, but this isn't really quadratic either. So maybe you want to model this as a X plus X squared plus maybe some function of sin of X or something. You actually sit around and fiddle with features. And after a while you can probably come up with a set of features that the model is okay, but let's talk about an algorithm that you can use without needing to do that.

So if – well, suppose you want to evaluate your hypothesis H at a certain point with a certain query point low K is X. Okay? And let's say you want to know what's the predicted value of Y at this position of X, right? So for linear regression, what we were doing was we would fit theta to minimize sum over I, YI minus theta, transpose XI squared, and return theta transpose X. Okay? So that was linear regression. In contrast, in locally weighted linear regression you're going to do things slightly different. You're going to look at this point X and then I'm going to look in my data set and take into account only the data points that are, sort of, in the little vicinity of X. Okay? So we'll look at where I want to value my hypothesis. I'm going to look only in the vicinity of this point where I want to value my hypothesis, and then I'm going to take, let's say, just these few points, and I will apply linear regression to fit a straight line just to this sub-set of the data. Okay? I'm using this sub-term sub-set – well let's come back to that later. So we take this data set and I fit a straight line to it and maybe I get a straight line like that. And what I'll do is then evaluate this particular value of straight line and that will be the value I return for my algorithm. I think this would be the predicted value for – this would be the value of then my hypothesis outputs in locally weighted regression. Okay?

So we're gonna fall one up. Let me go ahead and formalize that. In locally weighted regression, we're going to fit theta to minimize sum over I to minimize that where these terms W superscript I are called weights. There are many possible choice for ways, I'm just gonna write one down. So this E's and minus,

XI minus X squared over two. So let's look at what these weights really are, right? So notice that − suppose you have a training example XI. So that XI is very close to X. So this is small, right? Then if XI minus X is small, so if XI minus X is close to zero, then this is E's to the minus zero and E to the zero is one. So if XI is close to X, then WI will be close to one. In other words, the weight associated with the, I training example be close to one if XI and X are close to each other. Conversely if XI minus X is large then − I don't know, what would WI be?

Student: Zero.

Instructor (Andrew Ng): Zero, right. Close to zero. Right. So if XI is very far from X then this is E to the minus of some large number and E to the minus some large number will be close to zero. Okay? So the picture is, if I'm quarrying at a certain point X, shown on the X axis, and if my data set, say, look like that, then I'm going to give the points close to this a large weight and give the points far away a small weight. So for the points that are far away, WI will be close to zero. And so as if for the points that are far away, they will not contribute much at all to this summation, right? So I think this is sum over I of one times this quadratic term for points by points plus zero times this quadratic term for faraway points. And so the effect of using this weighting is that locally weighted linear regression fits a set of parameters theta, paying much more attention to fitting the points close by accurately. Whereas ignoring the contribution from faraway points. Okay? Yeah?

Student: Your Y is exponentially [inaudible]?

Instructor (Andrew Ng): Yeah. Let's see. So it turns out there are many other weighting functions you can use. It turns out that there are definitely different communities of researchers that tend to choose different choices by default. There is somewhat of a literature on debating what point − exactly what function to use. This, sort of, exponential decay function is − this happens to be a reasonably common one that seems to be a more reasonable choice on many problems, but you can actually plug in other functions as well. Did I mention what [inaudible] is it at? For those of you that are familiar with the normal distribution, or the Gaussian distribution, say this − what this formula I've written out here, it cosmetically looks a bit like a Gaussian distribution. Okay? But this actually has absolutely nothing to do with Gaussian distribution. So this is not that a problem with XI is Gaussian or whatever. This is no such interpretation. This is just a convenient function that happens to be a bell-shaped function, but don't endow this of any Gaussian semantics. Okay?

So, in fact − well, if you remember the familiar bell-shaped Gaussian, again, it's just the ways of associating with these points is that if you imagine putting this on a bell-shaped bump, centered around the position of where you want to value your hypothesis H, then there's a saying this point here I'll give a weight that's proportional to the height of the Gaussian − excuse me, to the height of the bell-shaped function evaluated at this point. And the way to get to this point will be, to this training example, will be proportionate to that height and so on. Okay? And so training examples that are really far away get a very small weight.

One last small generalization to this is that normally there's one other parameter to this algorithm, which I'll denote as tow. Again, this looks suspiciously like the variants of a Gaussian, but this is not a Gaussian. This is a convenient form or function. This parameter tow is called the bandwidth parameter and informally it controls how fast the weights fall of with distance. Okay? So just copy my diagram from the other side, I guess. So if tow is very small, if that's a query X, then you end up choosing a fairly narrow Gaussian − excuse me, a fairly narrow bell shape, so that the weights of the points are far away fall off rapidly. Whereas if tow is large then you'd end up choosing a weighting function that falls of relatively slowly with distance from your query. Okay?

So I hope you can, therefore, see that if you apply locally weighted linear regression to a data set that looks like this, then to ask what your hypothesis output is at a point like this you end up having a straight line making that prediction. To ask what kind of class this [inaudible] at that value you put a straight line there and you predict that value. It turns out that every time you try to vary your hypothesis, every time you ask your learning algorithm to make a prediction for how much a new house costs or whatever, you need to run a new fitting procedure and then evaluate this line that you fit just at the position of the value of X. So the position of the query where you're trying to make a prediction. Okay? But if you do this for every point along the X-axis then you find that locally weighted regression is able to trace on this, sort of, very

non-linear curve for a data set like this. Okay?

So in the problem set we're actually gonna let you play around more with this algorithm. So I won't say too much more about it here. But to finally move on to the next topic let me check the questions you have. Yeah?

Student: It seems like you still have the same problem of overfitting and underfitting, like when you had a Q's tow. Like you make it too small in your –

Instructor (Andrew Ng): Yes, absolutely. Yes. So locally weighted regression can run into – locally weighted regression is not a penancier for the problem of overfitting or underfitting. You can still run into the same problems with locally weighted regression. What you just said about – and so some of these things I'll leave you to discover for yourself in the homework problem. You'll actually see what you just mentioned. Yeah?

Student: It almost seems like you're not even thoroughly [inaudible] with this locally weighted, you had all the data that you originally had anyway.

Instructor (Andrew Ng): Yeah.

Student:I'm just trying to think of [inaudible] the original data points.

Instructor (Andrew Ng): Right. So the question is, sort of, this – it's almost as if you're not building a model, because you need the entire data set. And the other way of saying that is that this is a non-parametric learning algorithm. So this –I don't know. I won't debate whether, you know, are we really building a model or not. But this is a perfectly fine – so if I think when you write a code implementing locally weighted linear regression on the data set I think of that code as a whole – as building your model. So it actually uses – we've actually used this quite successfully to model, sort of, the dynamics of this autonomous helicopter this is. Yeah?

Student:I ask if this algorithm that learn the weights based on the data?

Instructor (Andrew Ng): Learn what weights? Oh, the weights WI.

Student: Instead of using [inaudible].

Instructor (Andrew Ng):I see, yes. So it turns out there are a few things you can do. One thing that is quite common is how to choose this band with parameter tow, right? As using the data. We'll actually talk about that a bit later when we talk about model selection. Yes? One last question.

Student:I used [inaudible] Gaussian sometimes if you [inaudible] Gaussian and then –

Instructor (Andrew Ng): Oh, I guess. Lt's see. Boy. The weights are not random variables and it's not, for the purpose of this algorithm, it is not useful to endow it with probable semantics. So you could choose to define things as Gaussian, but it, sort of, doesn't lead anywhere. In fact, it turns out that I happened to choose this, sort of, bell-shaped function to define my weights. It's actually fine to choose a function that doesn't even integrate to one, that integrates to infinity, say, as you're weighting function. So in that sense, I mean, you could force in the definition of a Gaussian, but it's, sort of, not useful. Especially since you use other functions that integrate to infinity and don't integrate to one. Okay? It's the last question and let's move on

Student: Assume that we have a very huge [inaudible], for example. A very huge set of houses and want to predict the linear for each house and so should the end result for each input – I'm seeing this very constantly for –

Instructor (Andrew Ng): Yes, you're right. So because locally weighted regression is a non-parametric algorithm every time you make a prediction you need to fit theta to your entire training set again. So you're actually right. If you have a very large training set then this is a somewhat expensive algorithm to use. Because every time you want to make a prediction you need to fit a straight line to a huge data set again. Turns out there are algorithms that – turns out there are ways to make this much more efficient for large data sets as well. So don't want to talk about that. If you're interested, look up the work of Andrew Moore on KD-trees. He, sort of, figured out ways to fit these models much more efficiently. That's not something I want to go into today. Okay? Let me move one. Let's take more questions later.

So, okay. So that's locally weighted regression. Remember the outline I had, I guess, at the beginning of this lecture. What I want to do now is talk about a probabilistic interpretation of linear regression, all right? And in particular of the – it'll be this probabilistic interpretation that let's us move on to talk about logistic

regression, which will be our first classification algorithm. So let's put aside locally weighted regression for now. We'll just talk about ordinary unweighted linear regression. Let's ask the question of why least squares, right? Of all the things we could optimize how do we come up with this criteria for minimizing the square of the area between the predictions of the hypotheses and the values Y predicted. So why not minimize the absolute value of the areas or the areas to the power of four or something? What I'm going to do now is present one set of assumptions that will serve to "justify" why we're minimizing the sum of square zero. Okay?

It turns out that there are many assumptions that are sufficient to justify why we do least squares and this is just one of them. So just because I present one set of assumptions under which least squares regression make sense, but this is not the only set of assumptions. So even if the assumptions I describe don't hold, least squares actually still makes sense in many circumstances. But this sort of new help, you know, give one rationalization, like, one reason for doing least squares regression. And, in particular, what I'm going to do is endow the least squares model with probabilistic semantics. So let's assume in our example of predicting housing prices, that the price of the house it's sold four, and there's going to be some linear function of the features, plus some term epsilon I. Okay? And epsilon I will be our error term. You can think of the error term as capturing unmodeled effects, like, that maybe there's some other features of a house, like, maybe how many fireplaces it has or whether there's a garden or whatever, that there are additional features that we jut fail to capture or you can think of epsilon as random noise. Epsilon is our error term that captures both these unmodeled effects. Just things we forgot to model. Maybe the function isn't quite linear or something. As well as random noise, like maybe that day the seller was in a really bad mood and so he sold it, just refused to go for a reasonable price or something. And now I will assume that the errors have a probabilistic – have a probability distribution. I'll assume that the errors epsilon I are distributed just till they denote epsilon I is distributive according to a probability distribution. That's a Gaussian distribution with mean zero and variance sigma squared. Okay? So let me just scripts in here, n stands for normal, right? To denote a normal distribution, also known as the Gaussian distribution, with mean zero and covariance sigma squared.

Actually, just quickly raise your hand if you've seen a Gaussian distribution before. Okay, cool. Most of you. Great. Almost everyone. So, in other words, the density for Gaussian is what you've seen before. The density for epsilon I would be one over root 2 pi sigma, E to the negative, epsilon I squared over 2 sigma squared, right? And the density of our epsilon I will be this bell-shaped curve with one standard deviation being a, sort of, sigma. Okay? This is form for that bell-shaped curve. So, let's see. I can erase that. Can I erase the board? So this implies that the probability distribution of a price of a house given in si and the parameters theta, that this is going to be Gaussian with that density. Okay? In other words, saying goes as that the price of a house given the features of the house and my parameters theta, this is going to be a random variable that's distributed Gaussian with mean theta transpose XI and variance sigma squared. Right? Because we imagine that the way the housing prices are generated is that the price of a house is equal to theta transpose XI and then plus some random Gaussian noise with variance sigma squared. So the price of a house is going to have mean theta transpose XI, again, and sigma squared, right? Does this make sense? Raise your hand if this makes sense. Yeah, okay. Lots of you.

In point of notation – oh, yes?

Student: Assuming we don't know anything about the error, why do you assume here the error is a Gaussian?

Instructor (Andrew Ng): Right. So, boy. Why do I see the error as Gaussian? Two reasons, right? One is that it turns out to be mathematically convenient to do so and the other is, I don't know, I can also mumble about justifications, such as things to the central limit theorem. It turns out that if you, for the vast majority of problems, if you apply a linear regression model like this and try to measure the distribution of the errors, not all the time, but very often you find that the errors really are Gaussian. That this Gaussian model is a good assumption for the error in regression problems like these. Some of you may have heard of the central limit theorem, which says that the sum of many independent random variables will tend towards a Gaussian. So if the error is caused by many effects, like the mood of the seller, the mood of the buyer, some other features that we miss, whether the place has a garden or not, and if all of these effects are independent,

then by the central limit theorem you might be inclined to believe that the sum of all these effects will be approximately Gaussian. If in practice, I guess, the two real answers are that, 1.) In practice this is actually a reasonably accurate assumption, and 2.) Is it turns out to be mathematically convenient to do so. Okay? Yeah?

Student: It seems like we're saying if we assume that area around model has zero mean, then the area is centered around our model. Which it seems almost like we're trying to assume what we're trying to prove. Instructor?

That's the [inaudible] but, yes. You are assuming that the error has zero mean. Which is, yeah, right. I think later this quarter we get to some of the other things, but for now just think of this as a mathematically – it's actually not an unreasonable assumption. I guess, in machine learning all the assumptions we make are almost never true in the absence sense, right? Because, for instance, housing prices are priced to dollars and cents, so the error will be – errors in prices are not continued as value random variables, because houses can only be priced at a certain number of dollars and a certain number of cents and you never have fractions of cents in housing prices. Whereas a Gaussian random variable would. So in that sense, assumptions we make are never "absolutely true," but for practical purposes this is a accurate enough assumption that it'll be useful to make. Okay? I think in a week or two, we'll actually come back to selected more about the assumptions we make and when they help our learning algorithms and when they hurt our learning algorithms. We'll say a bit more about it when we talk about generative and discriminative learning algorithms, like, in a week or two. Okay?

So let's point out one bit of notation, which is that when I wrote this down I actually wrote P of YI given XI and then semicolon theta and I'm going to use this notation when we are not thinking of theta as a random variable. So in statistics, though, sometimes it's called the frequentist's point of view, where you think of there as being some, sort of, true value of theta that's out there that's generating the data say, but we don't know what theta is, but theta is not a random vehicle, right? So it's not like there's some random value of theta out there. It's that theta is – there's some true value of theta out there. It's just that we don't know what the true value of theta is. So if theta is not a random variable, then I'm going to avoid writing P of YI given XI comma theta, because this would mean that probably of YI conditioned on X and theta and you can only condition on random variables. So at this part of the class where we're taking sort of frequentist's viewpoint rather than the Dasian viewpoint, in this part of class we're thinking of theta not as a random variable, but just as something we're trying to estimate and use the semicolon notation. So the way to read this is this is the probability of YI given XI and parameterized by theta. Okay? So you read the semicolon as parameterized by. And in the same way here, I'll say YI given XI parameterized by theta is distributed Gaussian with that.

All right. So we're gonna make one more assumption. Let's assume that the error terms are IID, okay? Which stands for Independently and Identically Distributed. So it's going to assume that the error terms are independent of each other, right? The identically distributed part just means that I'm assuming the outcome for the same Gaussian distribution or the same variance, but the more important part of is this is that I'm assuming that the epsilon I's are independent of each other. Now, let's talk about how to fit a model. The probability of Y given X parameterized by theta – I'm actually going to give this another name. I'm going to write this down and we'll call this the likelihood of theta as the probability of Y given X parameterized by theta. And so this is going to be the product over my training set like that. Which is, in turn, going to be a product of those Gaussian densities that I wrote down just now, right? Okay?

Then in parts of notation, I guess, I define this term here to be the likelihood of theta. And the likely of theta is just the probability of the data Y, right? Given X and prioritized by theta. To test the likelihood and probability are often confused. So the likelihood of theta is the same thing as the probability of the data you saw. So likely and probably are, sort of, the same thing. Except that when I use the term likelihood I'm trying to emphasize that I'm taking this thing and viewing it as a function of theta. Okay? So likelihood and for probability, they're really the same thing except that when I want to view this thing as a function of theta holding X and Y fix are then called likelihood. Okay? So hopefully you hear me say the likelihood of the parameters and the probability of the data, right? Rather than the likelihood of the data or probability of parameters. So try to be consistent in that terminology.

So given that the probability of the data is this and this is also the likelihood of the parameters, how do you estimate the parameters theta? So given a training set, what parameters theta do you want to choose for your model? Well, the principle of maximum likelihood estimation says that, right? You can choose the value of theta that makes the data as probable as possible, right? So choose theta to maximize the likelihood. Or in other words choose the parameters that make the data as probable as possible, right? So this is massive likely your estimation from six to six. So it's choose the parameters that makes it as likely as probable as possible for me to have seen the data I just did.

So for mathematical convenience, let me define lower case l of theta. This is called the log likelihood function and it's just log of capital L of theta. So this is log over product over I to find sigma E to that. I won't bother to write out what's in the exponent for now. It's just saying this from the previous board. Log and a product is the same as the sum of over logs, right? So it's a sum of the logs of – which simplifies to m times one over root two pi sigma plus and then log of explanation cancel each other, right? So if log of E of something is just whatever's inside the exponent. So, you know what, let me write this on the next board.

Okay. So maximizing the likelihood or maximizing the log likelihood is the same as minimizing that term over there. Well, you get it, right? Because there's a minus sign. So maximizing this because of the minus sign is the same as minimizing this as a function of theta. And this is, of course, just the same quadratic cos function that we had last time, J of theta, right? So what we've just shown is that the ordinary least squares algorithm, that we worked on the previous lecture, is just maximum likelihood assuming this probabilistic model, assuming IID Gaussian errors on our data. Okay?

One thing that we'll actually leave is that, in the next lecture notice that the value of sigma squared doesn't matter, right? That somehow no matter what the value of sigma squared is, I mean, sigma squared has to be a positive number. It's a variance of a Gaussian. So that no matter what sigma squared is since it's a positive number the value of theta we end up with will be the same, right? So because minimizing this you get the same value of theta no matter what sigma squared is. So it's as if in this model the value of sigma squared doesn't really matter. Just remember that for the next lecture. We'll come back to this again. Any questions about this? Actually, let me clean up another couple of boards and then I'll see what questions you have.

Okay. Any questions? Yeah?

Student: You are, I think here you try to measure the likelihood of your nice of theta by a fraction of error, but I think it's that you measure because it depends on the family of theta too, for example. If you have a lot of parameters [inaudible] or fitting in?

Instructor (Andrew Ng): Yeah, yeah. I mean, you're asking about overfitting, whether this is a good model. I think let's – the thing's you're mentioning are maybe deeper questions about learning algorithms that we'll just come back to later, so don't really want to get into that right now. Any more questions? Okay.

So this endows linear regression with a probabilistic interpretation. I'm actually going to use this probabil – use this, sort of, probabilistic interpretation in order to derive our next learning algorithm, which will be our first classification algorithm. Okay? So you'll recall that I said that regression problems are where the variable Y that you're trying to predict is continuous values. Now I'm actually gonna talk about our first classification problem, where the value Y you're trying to predict will be discreet value. You can take on only a small number of discrete values and in this case I'll talk about binding classification where Y takes on only two values, right? So you come up with classification problems if you're trying to do, say, a medical diagnosis and try to decide based on some features that the patient has a disease or does not have a disease. Or if in the housing example, maybe you're trying to decide will this house sell in the next six months or not and the answer is either yes or no. It'll either be sold in the next six months or it won't be. Other standing examples, if you want to build a spam filter. Is this e-mail spam or not? It's yes or no. Or if you, you know, some of my colleagues sit in whether predicting whether a computer system will crash. So you have a learning algorithm to predict will this computing cluster crash over the next 24 hours? And, again, it's a yes or no answer.

So there's X, there's Y. And in a classification problem Y takes on two values, zero and one. That's it in binding the classification. So what can you do? Well, one thing you could do is take linear regression, as

we've described it so far, and apply it to this problem, right? So you, you know, given this data set you can fit a straight line to it. Maybe you get that straight line, right? But this data set I've drawn, right? This is an amazingly easy classification problem. It's pretty obvious to all of us that, right? The relationship between X and Y is – well, you just look at a value around here and it's the right is one, it's the left and Y is zero. So you apply linear regressions to this data set and you get a reasonable fit and you can then maybe take your linear regression hypothesis to this straight line and threshold it at 0.5. If you do that you'll certainly get the right answer. You predict that if X is to the right of, sort of, the mid-point here then Y is one and then next to the left of that mid-point then Y is zero.

So some people actually do this. Apply linear regression to classification problems and sometimes it'll work okay, but in general it's actually a pretty bad idea to apply linear regression to classification problems like these and here's why. Let's say I change my training set by giving you just one more training example all the way up there, right? Imagine if given this training set is actually still entirely obvious what the relationship between X and Y is, right? It's just – take this value as greater than Y is one and it's less then Y is zero. By giving you this additional training example it really shouldn't change anything. I mean, I didn't really convey much new information. There's no surprise that this corresponds to Y equals one. But if you now fit linear regression to this data set you end up with a line that, I don't know, maybe looks like that, right? And now the predictions of your hypothesis have changed completely if your threshold – your hypothesis at Y equal both 0.5. Okay? So –

Student: In between there might be an interval where it's zero, right? For that far off point?

Instructor (Andrew Ng): Oh, you mean, like that?

Student: Right.

Instructor (Andrew Ng): Yeah, yeah, fine. Yeah, sure. A theta set like that so. So, I guess, these just – yes, you're right, but this is an example and this example works. This –

Student:[Inaudible] that will change it even more if you gave it all –

Instructor (Andrew Ng): Yeah. Then I think this actually would make it even worse. You would actually get a line that pulls out even further, right? So this is my example. I get to make it whatever I want, right? But the point of this is that there's not a deep meaning to this. The point of this is just that it could be a really bad idea to apply linear regression to classification algorithm. Sometimes it work fine, but usually I wouldn't do it. So a couple of problems with this. One is that, well – so what do you want to do for classification? If you know the value of Y lies between zero and one then to kind of fix this problem let's just start by changing the form of our hypothesis so that my hypothesis always lies in the unit interval between zero and one. Okay? So if I know Y is either zero or one then let's at least not have my hypothesis predict values much larger than one and much smaller than zero. And so I'm going to – instead of choosing a linear function for my hypothesis I'm going to choose something slightly different. And, in particular, I'm going to choose this function, H subscript theta of X is going to equal to G of theta transpose X where G is going to be this function and so this becomes more than one plus theta X of theta transpose X. And G of Z is called the sigmoid function and it is often also called the logistic function. It goes by either of these names.

And what G of Z looks like is the following. So when you have your horizontal axis I'm going to plot Z and so G of Z will look like this. Okay? I didn't draw that very well. Okay. So G of Z tends towards zero as Z becomes very small and G of Z will ascend towards one as Z becomes large and it crosses the vertical axis at 0.5. So this is what sigmoid function, also called the logistic function of. Yeah? Question?

Student: What sort of sigmoid in other step five?

Instructor (Andrew Ng): Say that again.

Student: Why we cannot chose this at five for some reason, like, that's better binary.

Instructor (Andrew Ng): Yeah. Let me come back to that later. So it turns out that Y – where did I get this function from, right? I just wrote down this function. It actually turns out that there are two reasons for using this function that we'll come to. One is – we talked about generalized linear models. We'll see that this falls out naturally as part of the broader class of models. And another reason that we'll talk about next week, it turns out there are a couple of, I think, very beautiful reasons for why we choose logistic functions. We'll see that in a little bit. But for now let me just define it and just take my word for it for

now that this is a reasonable choice. Okay? But notice now that my – the values output by my hypothesis will always be between zero and one. Furthermore, just like we did for linear regression, I'm going to endow the outputs and my hypothesis with a probabilistic interpretation, right? So I'm going to assume that the probability that Y is equal to one given X and parameterized by theta that's equal to H subscript theta of X, all right? So in other words I'm going to imagine that my hypothesis is outputting all these numbers that lie between zero and one. I'm going to think of my hypothesis as trying to estimate the probability that Y is equal to one. Okay? And because Y has to be either zero or one then the probability of Y equals zero is going to be that. All right? So more simply it turns out – actually, take these two equations and write them more compactly. Write P of Y given X parameterized by theta. This is going to be H subscript theta of X to the power of Y times one minus H of X to the power of one minus Y. Okay?

So I know this looks somewhat bizarre, but this actually makes the variation much nicer. So Y is equal to one then this equation is H of X to the power of one times something to the power of zero. So anything to the power of zero is just one, right? So Y equals one then this is something to the power of zero and so this is just one. So if Y equals one this is just saying P of Y equals one is equal to H subscript theta of X. Okay? And in the same way, if Y is equal to zero then this is P of Y equals zero equals this thing to the power of zero and so this disappears. This is just one times this thing power of one. Okay? So this is a compact way of writing both of these equations to gather them to one line. So let's hope our parameter fitting, right? And, again, you can ask – well, given this model by data, how do I fit the parameters theta of my model? So the likelihood of the parameters is, as before, it's just the probability of theta, right? Which is product over I, PFYI given XI parameterized by theta. Which is – just plugging those in. Okay? I dropped this theta subscript just so you can write a little bit less. Oh, excuse me. These should be XI's and YI's. Okay?

So, as before, let's say we want to find a maximum likelihood estimate of the parameters theta. So we want to find the – setting the parameters theta that maximizes the likelihood L of theta. It turns out that very often – just when you work with the derivations, it turns out that it is often much easier to maximize the log of the likelihood rather than maximize the likelihood. So the log likelihood L of theta is just log of capital L. This will, therefore, be sum of this. Okay? And so to fit the parameters theta of our model we'll find the value of theta that maximizes this log likelihood. Yeah?

Student:[Inaudible]

Instructor (Andrew Ng):Say that again.

Student:YI is [inaudible].

Instructor (Andrew Ng):Oh, yes. Thanks. So having maximized this function – well, it turns out we can actually apply the same gradient descent algorithm that we learned. That was the first algorithm we used to minimize the quadratic function. And you remember, when we talked about least squares, the first algorithm we used to minimize the quadratic error function was great in descent. So can actually use exactly the same algorithm to maximize the log likelihood. And you remember, that algorithm was just repeatedly take the value of theta and you replace it with the previous value of theta plus a learning rate alpha times the gradient of the cos function. The log likelihood will respect the theta. Okay? One small change is that because previously we were trying to minimize the quadratic error term. Today we're trying to maximize rather than minimize. So rather than having a minus sign we have a plus sign. So this is just great in ascents, but for the maximization rather than the minimization. So we actually call this gradient ascent and it's really the same algorithm.

So to figure out what this gradient – so in order to derive gradient descent, what you need to do is compute the partial derivatives of your objective function with respect to each of your parameters theta I, right? It turns out that if you actually compute this partial derivative – so you take this formula, this L of theta, which is – oh, got that wrong too. If you take this lower case l theta, if you take the log likelihood of theta, and if you take it's partial derivative with respect to theta I you find that this is equal to – let's see. Okay? And, I don't know, the derivation isn't terribly complicated, but in the interest of saving you watching me write down a couple of blackboards full of math I'll just write down the final answer. But the way you get this is you just take those, plug in the definition for F subscript theta as function of XI, and take derivatives, and work through the algebra it turns out it'll simplify down to this formula. Okay?

And so what that gives you is that gradient ascent is the following rule. Theta J gets updated as theta

J plus alpha gives this. Okay? Does this look familiar to anyone? Did you remember seeing this formula at the last lecture? Right. So when I worked up Bastrian descent for least squares regression I, actually, wrote down exactly the same thing, or maybe there's a minus sign and this is also fit. But I, actually, had exactly the same learning rule last time for least squares regression, right? Is this the same learning algorithm then? So what's different? How come I was making all that noise earlier about least squares regression being a bad idea for classification problems and then I did a bunch of math and I skipped some steps, but I'm, sort of, claiming at the end they're really the same learning algorithm?

Student:[Inaudible] constants?

Instructor (Andrew Ng): Say that again.

Student:[Inaudible]

Instructor (Andrew Ng): Oh, right. Okay, cool.

Student: It's the lowest it −

Instructor (Andrew Ng): No, exactly. Right. So zero to the same, this is not the same, right? And the reason is, in logistic regression this is different from before, right? The definition of this H subscript theta of XI is not the same as the definition I was using in the previous lecture. And in particular this is no longer theta transpose XI. This is not a linear function anymore. This is a logistic function of theta transpose XI. Okay? So even though this looks cosmetically similar, even though this is similar on the surface, to the Bastrian descent rule I derived last time for least squares regression this is actually a totally different learning algorithm. Okay? And it turns out that there's actually no coincidence that you ended up with the same learning rule. We'll actually talk a bit more about this later when we talk about generalized linear models. But this is one of the most elegant generalized learning models that we'll see later. That even though we're using a different model, you actually ended up with what looks like the same learning algorithm and it's actually no coincidence. Cool.

One last comment as part of a sort of learning process, over here I said I take the derivatives and I ended up with this line. I didn't want to make you sit through a long algebraic derivation, but later today or later this week, please, do go home and look at our lecture notes, where I wrote out the entirety of this derivation in full, and make sure you can follow every single step of how we take partial derivatives of this log likelihood to get this formula over here. Okay? By the way, for those who are interested in seriously masking machine learning material, when you go home and look at the lecture notes it will actually be very easy for most of you to look through the lecture notes and read through every line and go yep, that makes sense, that makes sense, that makes sense, and, sort of, say cool. I see how you get this line. You want to make sure you really understand the material. My concrete suggestion to you would be to you to go home, read through the lecture notes, check every line, and then to cover up the derivation and see if you can derive this example, right? So in general, that's usually good advice for studying technical material like machine learning. Which is if you work through a proof and you think you understood every line, the way to make sure you really understood it is to cover it up and see if you can rederive the entire thing itself. This is actually a great way because I did this a lot when I was trying to study various pieces of machine learning theory and various proofs. And this is actually a great way to study because cover up the derivations and see if you can do it yourself without looking at the original derivation. All right.

I probably won't get to Newton's Method today. I just want to say − take one quick digression to talk about one more algorithm, which was the discussion sort of alluding to this earlier, which is the perceptron algorithm, right? So I'm not gonna say a whole lot about the perceptron algorithm, but this is something that we'll come back to later. Later this quarter we'll talk about learning theory. So in logistic regression we said that G of Z are, sort of, my hypothesis output values that were low numbers between zero and one. The question is what if you want to force G of Z to up the value to either zero one? So the perceptron algorithm defines G of Z to be this. So the picture is − or the cartoon is, rather than this sigmoid function. E of Z now looks like this step function that you were asking about earlier. In saying this before, we can use H subscript theta of X equals G of theta transpose X. Okay? So this is actually − everything is exactly the same as before, except that G of Z is now the step function. It turns out there's this learning called the perceptron learning rule that's actually even the same as the classic gradient ascent for logistic regression. And the learning rule is given by this. Okay? So it looks just like the classic gradient ascent rule for logistic

regression. So this is very different flavor of algorithm than least squares regression and logistic regression, and, in particular, because it outputs only values are either zero or one it turns out it's very difficult to endow this algorithm with probabilistic semantics. And this is, again, even though – oh, excuse me. Right there. Okay. And even though this learning rule looks, again, looks cosmetically very similar to what we have in logistics regression this is actually a very different type of learning rule than the others that were seen in this class. So because this is such a simple learning algorithm, right? It just computes theta transpose X and then you threshold and then your output is zero or one. This is – right. So these are a simpler algorithm than logistic regression, I think. When we talk about learning theory later in this class, the simplicity of this algorithm will let us come back and use it as a building block. Okay? But that's all I want to say about this algorithm for now.

Just for fun, the last thing I'll do today is show you a historical video with – that talks about the perceptron algorithm. This particular video comes from a video series titled The Machine that Changed The World and was produced WGBH Television in cooperation with the BBC, British Broadcasting Corporation, and it aired on PBS a few years ago. This shows you what machine learning used to be like. It's a fun clip on perceptron algorithm.

In the 1950's and 60's scientists built a few working perceptrons, as these artificial brains were called. He's using it to explore the mysterious problem of how the brain learns. This perceptron is being trained to recognize the difference between males and females. It is something that all of us can do easily, but few of us can explain how. To get a computer to do this it would involve working out many complex rules about faces and writing a computer program, but this perceptron was simply given lots and lots of examples, including some with unusual hairstyles. But when it comes to a beetle the computer looks at facial features and hair outline and takes longer to learn what it's told by Dr. Taylor. Andrew puts on his wig also causes a little part searching. After training on lots of examples, it's given new faces it has never seen and is able to successfully distinguish male from female. It has learned.

All right. Isn't that great? Okay. That's it for today. I'll see you guys at the next lecture.
[End of Audio]
Duration: 75 minutes

1.4 Machine Learning Lecture 4[4]

Figure 1.4

Instructor (Andrew Ng):Okay, good morning. Just a few administrative announcements before we jump into today's technical material. So let's see, by later today, I'll post on the course website a handout with the sort of guidelines and suggestions for choosing and proposing class projects.

So project proposals – so for the term project for this class due on Friday, the 19th of this month at noon – that's about two weeks, two and a half weeks from now. If you haven't yet formed teams or started thinking about project ideas, please do so.

And later today, you'll find on the course website a handout with the guidelines and some of the details on how to send me your proposals and so on.

If you're not sure whether an idea you have for a project may be a appropriate, or you're sort of just fishing around for ideas or looking for ideas of projects to do, please, be strongly encouraged to come to my office hours on Friday mornings, or go to any of the TA's office hours to tell us about your project ideas, and we can help brainstorm with you.

---

[4]This content is available online at <http://cnx.org/content/m45870/1.3/>.

I also have a list of project ideas that I sort of collected from my colleagues and from various senior PhD students working with me or with other professors. And so if you want to hear about some of those ideas in topics like on natural [inaudible], computer vision, neuroscience, robotics, control. So [inaudible] ideas and a variety of topics at these, so if you're having trouble coming up with your own project idea, come to my office hours or to TA's office hours to ask us for suggestions, to brainstorm ideas with us.

Also, in the previous class I mentioned that we'll invite you to become [inaudible] with 229, which I think is a fun and educational thing to do. So later today, I'll also email everyone registered in this class with some of the logistical details about applying to be [inaudible]. So if you'd like to apply to be [inaudible], and I definitely encourage you to sort of consider doing so, please respond to that email, which you'll get later today.

And finally, problem set one will also be posted online shortly, and will be due in two weeks time, so you can also get that online.

Oh, and if you would like to be [inaudible], please try to submit problem set one on time and not use late days for problem set one because usually select [inaudible] is based on problem set one solutions. Questions for any of that?

Okay, so welcome back. And what I want to do today is talk about new test methods [inaudible] for fitting models like logistic regression, and then we'll talk about exponential family distributions and generalized linear models. It's a very nice class of ideas that will tie together, the logistic regression and the ordinary V squares models that we'll see. So hopefully I'll get to that today.

So throughout the previous lecture and this lecture, we're starting to use increasingly large amounts of material on probability. So if you'd like to see a refresher on sort of the foundations of probability – if you're not sure if you quite had your prerequisites for this class in terms of a background in probability and statistics, then the discussion section taught this week by the TA's will go over so they can review a probability.

At the same discussion sections also for the TA's, we'll also briefly go over sort of [inaudible] octave notation, which you need to use for your problem sets. And so if you any of you want to see a review of the probability and statistics pre-reqs, or if you want to [inaudible] octave, please come to this – the next discussion section.

All right. So just to recap briefly, towards the end of the last lecture I talked about the logistic regression model where we had – which was an algorithm for [inaudible]. We had that [inaudible] of [inaudible] – if an X – if Y equals one, give an X [inaudible] by theta under this model, all right. If this was one over one [inaudible] theta, transpose X. And then you can write down the log like we heard – like given the training sets, which was that.

And by taking the riveters of this, you can derive sort of a gradient ascent interval for finding the maximum likelihood estimate of the parameter stated for this logistic regression model.

And so last time I wrote down the learning rule for [inaudible], but the [inaudible] has to be gradient ascent where you look at just one training example at a time, would be like this, okay. So last time I wrote down [inaudible] gradient ascent. This is still [inaudible] gradient ascent.

So if you want to favor a logistic regression model, meaning find the value of theta that maximizes this log likelihood, gradient ascent or [inaudible] gradient ascent or [inaudible] gradient ascent is a perfectly fine algorithm to use.

But what I want to do is talk about a different algorithm for fitting models like logistic regression. And this would be an algorithm that will, I guess, often run much faster than gradient ascent.

And this algorithm is called Newton's Method. And when we describe Newton's Method – let me ask you – I'm actually going to ask you to consider a different problem first, which is – let's say you have a function F of theta, and let's say you want to find the value of theta so that F of theta is equal to zero.

Let's start the [inaudible], and then we'll sort of slowly change this until it becomes an algorithm for fitting mass and likelihood models, like [inaudible] reduction.

So – let's see. I guess that works. Okay, so let's say that's my function F. This is my horizontal axis of [inaudible] of theta, and so they're really trying to find this value for theta, and which F of theta is equal to zero. This is a horizontal axis.

So here's the [inaudible]. I'm going to initialize theta as some value. We'll call theta superscript zero. And then here's what Newton's Method does. We're going to evaluate the function F at a value of theta, and then we'll compute it over to the [inaudible], and we'll use the linear consummation to the function F of that value of theta. So in particular, I'm going to take the tangents to my function – hope that makes sense – starting the function [inaudible] work out nicely.

I'm going to take the tangent to my function at that point there to zero, and I'm going to sort of extend this tangent down until it intercepts the horizontal axis. I want to see what value this is. And I'm going to call this theta one, okay. And then so that's one iteration of Newton's Method.

And what I'll do then is the same thing with the dec point. Take the tangent down here, and that's two iterations of the algorithm. And then just sort of keep going, that's theta three and so on, okay.

So let's just go ahead and write down what this algorithm actually does. To go from theta zero to theta one, let me call that length – let me just call that capital delta.

So capital – so if you remember the definition of a derivative [inaudible], derivative of F evaluated at theta zero. In other words, the gradient of this first line, by the definition of gradient is going to be equal to this vertical length, divided by this horizontal length. A gradient of this – so the slope of this function is defined as the ratio between this vertical height and this width of triangle.

So that's just equal to F of theta zero, divided by delta, which implies that delta is equal to F of theta zero, divided by a prime of theta zero, okay.

And so theta one is therefore theta zero minus delta, minus capital delta, which is therefore just F theta zero over F prime of theta zero, all right.

And more generally, one iteration of Newton's Method precedes this, theta T plus one equals theta T minus F of theta T divided by F prime of theta T. So that's one iteration of Newton's Method.

Now, this is an algorithm for finding a value of theta for which F of theta equals zero. And so we apply the same idea to maximizing the log likelihood, right. So we have a function L of theta, and we ant to maximize this function.

Well, how do you maximize the function? You set the derivative to zero. So we want theta [inaudible]. Our prime of theta is equal to zero, so to maximize this function we want to find the place where the derivative of the function is equal to zero, and so we just apply the same idea. So we get theta one equals theta T minus L prime of theta T over L double prime of T, L double prime of theta T, okay.

Because to maximize this function, we just let F be equal to L prime. Let F be the [inaudible] of L, and then we want to find the value of theta for which the derivative of L is zero, and therefore must be a local optimum. Does this make sense? Any questions about this?

Student:[Inaudible]

Instructor (Andrew Ng): The answer to that is fairly complicated. There are conditions on F that would guarantee that this will work. They are fairly complicated, and this is more complex than I want to go into now. In practice, this works very well for logistic regression, and for sort of generalizing any models I'll talk about later.

Student:[Inaudible]

Instructor (Andrew Ng): Yeah, it usually doesn't matter. When I implement this, I usually just initialize theta zero to zero to just initialize the parameters to the – back to all zeros, and usually this works fine. It's usually not a huge deal how you initialize theta.

Student:[Inaudible] or is it just different conversions?

Instructor (Andrew Ng): Let me say some things about that that'll sort of answer it. All of these algorithms tend not to – converges problems, and all of these algorithms will generally converge, unless you choose too large a linear rate for gradient ascent or something. But the speeds of conversions of these algorithms are very different.

So it turns out that Newton's Method is an algorithm that enjoys extremely fast conversions. The technical term is that it enjoys a property called [inaudible] conversions. Don't know [inaudible] what that means, but just stated informally, it means that [inaudible] every iteration of Newton's Method will double the number of significant digits that your solution is accurate to. Just lots of constant factors.

Suppose that on a certain iteration your solution is within 0.01 at the optimum, so you have 0.01 error.

Then after one iteration, your error will be on the order of 0.001, and after another iteration, your error will be on the order of 0.0001. So this is called [inaudible] conversions because you essentially get to square the error on every iteration of Newton's Method.

[Inaudible] result that holds only when your [inaudible] cause the optimum anyway, so this is the theoretical result that says it's true, but because of constant factors and so on, may paint a slightly rosier picture than might be accurate.

But the fact is, when you implement – when I implement Newton's Method for logistic regression, usually converges like a dozen iterations or so for most reasonable size problems of tens of hundreds of features.

So one thing I should talk about, which is what I wrote down over there was actually Newton's Method for the case of theta being a single-row number. The generalization to Newton's Method for when theta is a vector rather than when theta is just a row number is the following, which is that theta T plus one is theta T plus – and then we have the second derivative divided by the first – the first derivative divided by the second derivative.

And the appropriate generalization is this, where this is the usual gradient of your objective, and each [inaudible] is a matrix called a Hessian, which is just a matrix of second derivative where HIJ equals – okay.

So just to sort of – the first derivative divided by the second derivative, now you have a vector of first derivatives times sort of the inverse of the matrix of second derivatives. So this is sort of just the same thing [inaudible] of multiple dimensions.

So for logistic regression, again, use the – for a reasonable number of features and training examples – when I run this algorithm, usually you see a conversion anywhere from sort of [inaudible] to like a dozen or so other [inaudible].

To compare to gradient ascent, it's [inaudible] to gradient ascent, this usually means far fewer iterations to converge. Compared to gradient ascent, let's say [inaudible] gradient ascent, the disadvantage of Newton's Method is that on every iteration you need to invert the Hessian.

So the Hessian will be an N-by-N matrix, or an N plus one by N plus one-dimensional matrix if N is the number of features. And so if you have a large number of features in your learning problem, if you have tens of thousands of features, then inverting H could be a slightly computationally expensive step. But for smaller, more reasonable numbers of features, this is usually a very [inaudible]. Question?

Student:[Inaudible]

Instructor (Andrew Ng): Let's see. I think you're right. That should probably be a minus. Do you have [inaudible]? Yeah, thanks. Yeah, X to a minus.

Thank you. [Inaudible] problem also. I wrote down this algorithm to find the maximum likely estimate of the parameters for logistic regression. I wrote this down for maximizing a function. So I'll leave you to think about this yourself.

If I wanted to use Newton's Method to minimize the function, how does the algorithm change? All right. So I'll leave you to think about that. So in other words, it's not the maximizations. How does the algorithm change if you want to use it for minimization? Actually, the answer is that it doesn't change. I'll leave you to work that out yourself why, okay.

All right. Let's talk about generalized linear models. Let me just say, just to give a recap of both of the algorithms we've talked about so far. We've talked about two different algorithms for modeling PFY given X and parameterized by theta.

And one of them – R was a real number and we are sealing that. And we sort of – the [inaudible] has a Gaussian distribution, then we got [inaudible] of linear regression.

In the other case, we saw that if – was a classification problem where Y took on a value of either zero or one. In that case, well, what's the most natural distribution of zeros and ones is the [inaudible]. The [inaudible] distribution models random variables with two values, and in that case we got logistic regression.

So along the way, some of the questions that came up were – so logistic regression, where on earth did I get the [inaudible] function from? And then so there are the choices you can use for, sort of, just where did this function come from?

And there are other functions I could've plugged in, but the [inaudible] function turns out to be a natural default choice that lead us to logistic regression. And what I want to do now is take both of these algorithms

and show that there are special cases that have [inaudible] the course of algorithms called generalized linear models, and there will be pauses for − it will be as [inaudible] the course of algorithms that think that the [inaudible] function will fall out very naturally as well.

So, let's see − just looking for a longer piece of chalk. I should warn you, the ideas in generalized linear models are somewhat complex, so what I'm going to do today is try to sort of point you − point out the key ideas and give you a gist of the entire story. And then some of the details in the map and the derivations I'll leave you to work through by yourselves in the intellection [inaudible], which posts online.

So [inaudible] these two distributions, the [inaudible] and the Gaussian. So suppose we have data that is zero-one valued, and we and we want to model it with [inaudible] variable parameterized by phi. So the [inaudible] distribution has the probability of Y equals one, which just equals the phi, right. So the parameter phi in the [inaudible] specifies the probability of Y being one.

Now, as you vary the parameter theta, you get − you sort of get different [inaudible] distributions. As you vary the value of theta you get different probability distributions on Y that have different probabilities of being equal to one. And so I want you to think of this as not one fixed distribution, but as a set where there are a cause of distributions that you get as you vary theta.

And in the same way, if you consider Gaussian distribution, as you vary [inaudible] you would get different Gaussian distributions. So think of this again as a cost, or as a set to distributions.

And what I want to do now is show that both of these are special cases of the cause of distribution that's called the exponential family distribution. And in particular, we'll say that the cost of distributions, like the [inaudible] distributions that you get as you vary theta, we'll say the cost of distributions is in the exponential family if it can be written in the following form. P of Y parameterized by theta is equal to B of Y [inaudible], okay.

Let me just get some of these terms, names, and then − let me − I'll say a bit more about what this means. So [inaudible] is called the natural parameter of the distribution, and T of Y is called the sufficient statistic. Usually, for many of the examples we'll see, including the [inaudible] and the Gaussian, T of Y is just equal to Y. So for most of this lecture you can mentally replace T of Y to be equal to Y, although this won't be true for the very fine example we do today, but mentally, you think of T of Y as equal to Y.

And so for a given choice of these functions, A, B and T, all right − so we're gonna sort of fix the forms of the functions A, B and T. Then this formula defines, again, a set of distributions. It defines the cause of distributions that is now parameterized by [inaudible].

So again, let's write down specific formulas for A, B and T, true specific choices of A, B and T. Then as I vary [inaudible] I get different distributions. And I'm going to show that the [inaudible] − I'm going to show that the [inaudible] and the Gaussians are special cases of exponential family distributions. And by that I mean that I can choose specific functions, A, B and T, so that this becomes the formula of the distributions of either a [inaudible] or a Gaussian.

And then again, as I vary [inaudible], I'll get [inaudible], distributions with different means, or as I vary [inaudible], I'll get Gaussian distributions with different means for my fixed values of A, B and T.

And for those of you that know what a sufficient statistic and statistics is, T of Y actually is a sufficient statistic in the formal sense of sufficient statistic for a probability distribution. They may have seen it in a statistics class. If you don't know what a sufficient statistic is, don't worry about. We sort of don't need that property today.

Okay. So − oh, one last comment. Often, T of Y is equal to Y, and in many of these cases, [inaudible] is also just a raw number. So in many cases, the parameter of this distribution is just a raw number, and [inaudible] transposed T of Y is just a product of raw numbers. So again, that would be true for our first two examples, but now for the last example I'll do today.

So now we'll show that the [inaudible] and the Gaussian are examples of exponential family distributions. We'll start with the [inaudible]. So the [inaudible] distribution with [inaudible] − I guess I wrote this down already. PFY equals one [inaudible] by phi, [inaudible] equal to phi. So the parameter of phi specifies the probability that Y equals one.

And so my goal now is to choose T, A and B, or is to choose A, B and T so that my formula for the exponential family becomes identical to my formula for the distribution of a [inaudible].

So probability of Y parameterized by phi is equal to that, all right. And you already saw sort of a similar exponential notation where we talked about logistic regression. The probability of Y being one is phi, the probability of Y being zero is one minus phi, so we can write this compactly as phi to the Y times one minus phi to the one minus Y.

So I'm gonna take the exponent of the log of this, an exponentiation in taking log [inaudible] cancel each other out [inaudible]. And this is equal to E to the Y. And so [inaudible] is to be T of Y, and this will be minus A of [inaudible]. And then B of Y is just one, so B of Y doesn't matter.

Just take a second to look through this and make sure it makes sense. I'll clean another board while you do that.

So now let's write down a few more things. Just copying from the previous board, we had that [inaudible] zero four equal to log phi over one minus phi.

[Inaudible] so if I want to do the [inaudible] take this formula, and if you invert it, if you solve for phi – excuse me, if you solve for theta as a function of phi, which is really [inaudible] is the function of phi. Just invert this formula. You find that phi is one over one plus [inaudible] minus [inaudible]. And so somehow the logistic function magically falls out of this. We'll take this even this even further later.

Again, copying definitions from the board on – from the previous board, A of [inaudible] I said is minus log of one minus phi. So again, phi and [inaudible] are function of each other, all right. So [inaudible] depends on phi, and phi depends on [inaudible].

So if I plug in this definition for [inaudible] into this – excuse me, plug in this definition for phi into that, I'll find that A of [inaudible] is therefore equal to log one plus [inaudible] to [inaudible]. And again, this is just algebra. This is not terribly interesting. And just to complete – excuse me. And just to complete the rest of this, T of Y is equal to Y, and B of Y is equal to one, okay.

So just to recap what we've done, we've come up with a certain choice of functions A, T and B, so then my formula for the exponential family distribution now becomes exactly the formula for the distributions, or for the probability mass function of the [inaudible] distribution. And the natural parameter [inaudible] has a certain relationship of the original parameter of the [inaudible]. Question?

Student:[Inaudible]

Instructor (Andrew Ng):Let's see. [Inaudible].

Student: The second to the last one.

Instructor (Andrew Ng):Oh, this answer is fine.

Student: Okay.

Instructor (Andrew Ng):Let's see. Yeah, so this is – well, if you expand this term out, one minus Y times log Y minus phi, and so one times log – one minus phi becomes this. And the other term is minus Y times log Y minus phi. And then – so the minus of a log is log one over X, or is just log one over whatever. So minus Y times log one minus phi becomes sort of Y times log, one over one minus phi. Does that make sense?

Student: Yeah.

Instructor (Andrew Ng):Yeah, cool. Anything else? Yes?

Student:[Inaudible] is a scaler, isn't it? Up there –

Instructor (Andrew Ng):Yes.

Student:– it's a [inaudible] transposed, so it can be a vector or –

Instructor (Andrew Ng):Yes, [inaudible]. So let's see. In most – in this and the next example, [inaudible] will turn out to be scaler. And so – well, on this board. And so if [inaudible] is a scaler and T of Y is a scaler, then this is just a real number times a real number. So this would be like a one-dimensional vector transposed times a one-dimensional vector. And so this is just real number times real number.

Towards the end of today's lecture, we'll go with just one example where both of these are vectors. But for main distributions, these will turn out to be scalers.

Student:[Inaudible] distribution [inaudible]. I mean, it doesn't have the zero probability or [inaudible] zero and one.

Instructor (Andrew Ng):I see. So – yeah. Let's – for this, let's imagine that we're restricting the domain of the input of the function to be Y equals zero or one. So think of that as maybe in implicit

constraint on it. [Inaudible]. But so this is a probability mass function for Y equals zero or Y equals one. So write down Y equals zero one. Let's think of that as an [inaudible].

So – cool. So this takes the [inaudible] distribution and invites in the form and the exponential family distribution. [Inaudible] do that very quickly for the Gaussian. I won't do the algebra for the Gaussian. I'll basically just write out the answers.

So with a normal distribution with [inaudible] sequence squared, and so you remember, was it two lectures ago, when we were dividing the maximum likelihood – excuse me, oh, no, just the previous lecture when we were dividing the maximum likelihood estimate for the parameters of ordinary [inaudible] squares. We showed that the parameter for [inaudible] squared didn't matter.

When we divide the [inaudible] model for [inaudible] square [inaudible], we said that no matter what [inaudible] square was, we end up with the same value of the parameters.

So for the purposes of just writing lesson, today's lecture, and not taking account [inaudible] squared, I'm just going to set [inaudible] squared to be for the one, okay, so as to not worry about it.

Lecture [inaudible] talks a little bit more about this, but I'm just gonna – just to make [inaudible] in class a bit easier and simpler today, let's just say that [inaudible] square equals one. [Inaudible] square is essentially just a scaling factor on the variable Y.

So in that case, the Gaussian density is given by this, [inaudible] squared. And – well, by a couple of steps of algebra, which I'm not going to do, but is written out in [inaudible] in the lecture now so you can download. This is one root two pie, E to the minus one-half Y squared times E to E. New Y minus one-half [inaudible] squared, okay. So I'm just not doing the algebra.

And so that's B of Y, we have [inaudible] that's equal to [inaudible]. P of Y equals Y, and – well, A of [inaudible] is equal to minus one-half – actually, I think that should be plus one-half. Have I got that right? Yeah, sorry. Let's see – excuse me. Plus sign there, okay. If you minus one-half [inaudible] squared, and because [inaudible] is equal to [inaudible], this is just minus one-half [inaudible] squared, okay.

And so this would be a specific choice again of A, B and T that expresses the Gaussian density in the form of an exponential family distribution. And in this case, the relationship between [inaudible] and [inaudible] is that [inaudible] is just equal to [inaudible], so the [inaudible] of the Gaussian is just equal to the natural parameter of the exponential family distribution.

Student: Minus half.

Instructor (Andrew Ng):Oh, this is minus half?

Student:[Inaudible]

Instructor (Andrew Ng):Oh, okay, thanks. And so – guessing that should be plus then. Is that right? Okay. Oh, yes, you're right. Thank you. All right.

And so [inaudible] result that if you've taken a look in undergrad statistics class, turns out that most of the "textbook distributions," not all, but most of them, can be written in the form of an exponential family distribution.

So you saw the Gaussian, the normal distribution. It turns out the [inaudible] in normal distribution, which is a generalization of Gaussian random variables, so it's a high dimension to vectors. The [inaudible] normal distribution is also in the exponential family.

You saw the [inaudible] as an exponential family. It turns out the [inaudible] distribution is too, all right. So the [inaudible] models outcomes over zero and one. They'll be coin tosses with two outcomes. The [inaudible] models outcomes over K possible values. That's also an exponential families distribution.

You may have heard of the Parson distribution. And so the Parson distribution is often used for modeling counts. Things like the number of radioactive decays in a sample, or the number of customers to your website, the numbers of visitors arriving in a store. The Parson distribution is also in the exponential family.

So are the gamma and the exponential distributions, if you've heard of them. So the gamma and the exponential distributions are distributions of the positive numbers. So they're often used in model intervals, like if you're standing at the bus stop and you want to ask, "When is the next bus likely to arrive? How long do I have to wait for my bus to arrive?" Often you model that with sort of gamma distribution or exponential families, or the exponential distribution. Those are also in the exponential family.

Even more [inaudible] distributions, like the [inaudible] and the [inaudible] distributions, these are probably distributions over fractions, are already probability distributions over probability distributions. And also things like the Wisha distribution, which is the distribution over covariance matrices. So all of these, it turns out, can be written in the form of exponential family distributions.

Well, and in the problem set where he asks you to take one of these distributions and write it in the form of the exponential family distribution, and derive a generalized linear model for it, okay.

Which brings me to the next topic of having chosen and exponential family distribution, how do you use it to derive a generalized linear model? So generalized linear models are often abbreviated GLM's. And I'm going to write down the three assumptions. You can think of them as assumptions, or you can think of them as design choices, that will then allow me to sort of turn a crank and come up with a generalized linear model.

So the first one is – I'm going to assume that given my input X and my parameters theta, I'm going to assume that the variable Y, the output Y, or the response variable Y I'm trying to predict is distributed exponential family with some natural parameter [inaudible].

And so this means that there is some specific choice of those functions, A, B and T so that the conditional distribution of Y given X and parameterized by theta, those exponential families with parameter [inaudible]. Where here, [inaudible] may depend on X in some way.

So for example, if you're trying to predict – if you want to predict how many customers have arrived at your website, you may choose to model the number of people – the number of hits on your website by Parson Distribution since Parson Distribution is natural for modeling com data. And so you may choose the exponential family distribution here to be the Parson distribution.

[Inaudible] that given X, our goal is to output the effective value of Y given X. So given the features in the website examples, I've given a set of features about whether there were any proportions, whether there were sales, how many people linked to your website, or whatever. I'm going to assume that our goal in our [inaudible] problem is to estimate the expected number of people that will arrive at your website on a given day.

So in other words, you're saying that I want H of X to be equal to – oh, excuse me. I actually meant to write T of Y here. My goal is to get my learning algorithms hypothesis to output the expected value of T of Y given X.

But again, for most of the examples, T of Y is just equal to Y. And so for most of the examples, our goal is to get our learning algorithms output, T expected value of Y given X because T of Y is usually equal to Y. Yes?

Student:[Inaudible]

Instructor (Andrew Ng): Yes, same thing, right. T of Y is a sufficient statistic. Same T of Y.

And lastly, this last one I wrote down – these are assumptions. This last one you might – maybe wanna think of this as a design choice. Which is [inaudible] assume that the distribution of Y given X is a distributed exponential family with some parameter [inaudible].

So the number of visitors on the website on any given day will be Parson or some parameter [inaudible]. And the last decision I need to make is was the relationship between my input teachers and this parameter [inaudible] parameterizing my Parson distribution or whatever.

And this last step, I'm going to make the assumption, or really a design choice, that I'm going to assume the relationship between [inaudible] and my [inaudible] axis linear, and in particular that they're governed by this – that [inaudible] is equal to theta, transpose X.

And the reason I make this design choice is it will allow me to turn the crank of the generalized linear model of machinery and come off with very nice algorithms for fitting say Parson Regression models or performed regression with a gamma distribution outputs or exponential distribution outputs and so on.

So let's work through an example. [Inaudible] equals theta transpose X works for the case where [inaudible] is a real number. For the more general case, you would have [inaudible] I equals theta I, transpose X if [inaudible] is a vector rather than a real number. But again, most of the examples [inaudible] will just be a real number.

All right. So let's work through the [inaudible] example. You'll see where Y given X parameterized by

theta – this is a distributed exponential family with natural parameter [inaudible]. And for the [inaudible] distribution, I'm going to choose A, B and T to be the specific forms that cause those exponential families to become the [inaudible] distribution. This is the example we worked through just now, the first example we worked through just now.

So – oh, and we also have – so for any fixed value of X and theta, my hypothesis, my learning algorithm will make a prediction, or will make – will sort of output [inaudible] of X, which is by my, I guess, assumption [inaudible].

Watch our learning algorithm to output the expected value of Y given X and parameterized by theta, where Y can take on only the value zero and one, then the expected value of Y is just equal to the probability that Y is equal to one. So the expected value of a [inaudible] variable is just equal to the probability that it's equal to one.

And so the probability that Y equals one is just equal to phi because that's the parameter of my [inaudible] distribution. Phi is, by definition, I guess, is the probability of my [inaudible] distribution [inaudible] value of one.

Which we worked out previously, phi was one over one plus E to the negative [inaudible]. So we worked this out on our previous board. This is the relationship – so when we wrote down the [inaudible] distribution in the form of an exponential family, we worked out what the relationship was between phi and [inaudible], and it was this. So we worked out the relationship between the expected value of Y and [inaudible] was this relationship.

And lastly, because we made the design choice, or the assumption that [inaudible] and theta are linearly related. This is therefore equal to one over one plus E to the minus theta, transpose X.

And so that's how I come up with the logistic regression algorithm when you have a variable Y – when you have a [inaudible] variable Y, or also response variable Y that takes on two values, and then you choose to model variable [inaudible] distribution. Are you sure this does make sense? Raise your hand if this makes sense. Yeah, okay, cool.

So I hope you get the ease of use of this, or sort of the power of this. The only decision I made was really, I said Y – let's say I have a new machine-learning problem and I'm trying to predict the value of a variable Y that happens to take on two values. Then the only decision I need to make is I chose [inaudible] distribution.

I say I want to model – I want to assume that given X and theta, I'm going to assume Y is distributed [inaudible]. That's the only decision I made. And then everything else follows automatically having made the decision to model Y given X and parameterized by theta as being [inaudible].

In the same way you can choose a different distribution, you can choose Y as Parson or Y as gamma or Y as whatever, and follow a similar process and come up with a different model and different learning algorithm. Come up with a different generalized linear model for whatever learning algorithm you're faced with.

This tiny little notation, the function G that relates G of [inaudible] that relates the natural parameter to the expected value of Y, which in this case, one over one plus [inaudible] minus [inaudible], this is called the canonical response function. And G inverse is called the canonical link function.

These aren't a huge deal. I won't use this terminology a lot. I'm just mentioning those in case you hear about – people talk about generalized linear models, and if they talk about canonical response functions or canonical link functions, just so you know there's all of this.

Actually, many techs actually use the reverse way. This is G inverse and this is G, but this notation turns out to be more consistent with other algorithms in machine learning. So I'm going to use this notation. But I probably won't use the terms canonical response functions and canonical link functions in lecture a lot, so just – I don't know. I'm not big on memorizing lots of names of things. I'm just tossing those out there in case you see it elsewhere.

Okay. You know what, I think in the interest of time, I'm going to skip over the Gaussian example. But again, just like I said, [inaudible], Y is [inaudible], different variation I get of logistic regression. You can do the same thing with the Gaussian distribution and end up with ordinary [inaudible] squares model.

The problem with Gaussian is that it's almost so simple that when you see it for the first time that it's

sometimes more confusing than the [inaudible] model because it looks so simple, it looks like it has to be more complicated. So let me just skip that and leave you to read about the Gaussian example in the lecture notes.

And what I want to do is actually go through a more complex example. Question?

Student:[Inaudible]

Instructor (Andrew Ng):Okay, right. So how do choose what theory will be? We'll get to that in the end. What you have there is the logistic regression model, which is a [inaudible] model that assumes the probability of Y given X is given by a certain form.

And so what you do is you can write down the log likelihood of your training set, and find the value of theta that maximizes the log likelihood of the parameters. Does that make sense? So I'll say that again towards the end of today's lecture.

But for logistic regression, the way you choose theta is exactly maximum likelihood, as we worked out in the previous lecture, using Newton's Method or gradient ascent or whatever. I'll sort of try to do that again for one more example towards the end of today's lecture.

So what I want to do is actually use the remaining, I don't know, 19 minutes or so of this class, to go through the – one of the more – it's probably the most complex example of a generalized linear model that I've used. This one I want to go through because it's a little bit trickier than many of the other textbook examples of generalized linear models.

So again, what I'm going to do is go through the derivation reasonably quickly and give you the gist of it, and if there are steps I skip or details omitted, I'll leave you to read about them more carefully in the lecture notes.

And what I want to do is talk about [inaudible]. And [inaudible] is the distribution over K possible outcomes. Imagine you're now in a machine-learning problem where the value of Y that you're trying to predict can take on K possible outcomes, so rather than only two outcomes.

So obviously, this example's already – if you want to have a learning algorithm, or to magically send emails for you into your right email folder, and you may have a dozen email folders you want your algorithm to classify emails into. Or predicting if the patient either has a disease or does not have a disease, which would be a [inaudible] classification problem.

If you think that the patient may have one of K diseases, and you want other than have a learning algorithm figure out which one of K diseases your patient has is all.

So lots of multi-cause classification problems where you have more than two causes. You model that with [inaudible]. And eventually – so for logistic regression, I had [inaudible] like these where you have a training set and you find a decision boundary that separates them.

[Inaudible], we're going to entertain the value of predicting, taking on multiple values, so you now have three causes, and the learning algorithm will learn some way to separate out three causes or more, rather than just two causes.

So let's write [inaudible] in the form of an exponential family distribution. So the parameters of a [inaudible] are phi one, phi two [inaudible] phi K. I'll actually change this in a second – where the probability of Y equals I is phi I, right, because there are K possible outcomes.

But if I choose this as my parameterization of the [inaudible], then my parameter's actually redundant because if these are probabilities, then you have to sum up the one. And therefore for example, I can derive the last parameter, phi K, as one minus phi one, up to phi K minus one. So this would be a redundant parameterization from [inaudible]. The result is over-parameterized.

And so for purposes of this [inaudible], I'm going to treat my parameters of my [inaudible] as phi one, phi two, up to phi K minus one. And I won't think of phi K as a parameter. I'll just – so my parameters are just – I just have K minus one parameters, parameterizing my [inaudible].

And sometimes I write phi K in my derivations as well, and you should think of phi K as just a shorthand for this, for one minus the rest of the parameters, okay.

So it turns out the [inaudible] is one of the few examples where T of Y – it's one of the examples where T of Y is not equal to Y. So in this case, Y is on of K possible values.

And so T of Y would be defined as follows; T of one is going to be a vector with a one and zeros

everywhere else. T of two is going to be a zero, one, zero and so on. Except that these are going to be K minus one-dimensional vectors.

And so T of K minus one is going to be zero, zero, zero, one. And T of K is going to be the vector of all zeros. So this is just how I'm choosing to define T of Y to write down the [inaudible] in the form of an exponential family distribution. Again, these are K minus one-dimensional vectors.

So this is a good point to introduce one more useful piece of notation, which is called indicator function notation. So I'm going to write one, and then curly braces. And if I write a true statement inside, then the indicator of that statement is going to be one. Then I write one, and then I write a false statement inside, then the value of this indicator function is going to be a zero.

For example, if I write indicator two equals three [inaudible] that's false, and so this is equal to zero. Whereas indicator [inaudible] plus one equals two, I wrote down a true statement inside. And so the indicator of the statement was equal to one. So the indicator function is just a very useful notation for indicating sort of truth or falsehood of the statement inside.

And so − actually, let's do this here. To combine both of these, right, if I carve out a bit of space here − so if I use − so TY is a vector. Y is one of K values, and so TY is one of these K vectors. If I use TY as [inaudible] to denote the [inaudible] element of the vector TY, then TY − the [inaudible] element of the vector TY is just equal to indicator for whether Y is equal to I.

Just take a − let me clean a couple more boards. Take a look at this for a second and make sure you understand why that − make sure you understand all that notation and why this is true.

All right. Actually, raise your hand if this equation makes sense to you. Most of you, not all, okay. [Inaudible].

Just as one kind of [inaudible], suppose Y is equal to one − let's say − let me see. Suppose Y is equal to one, right, so TY is equal to this vector, and therefore the first element of this vector will be one, and the rest of the elements will be equal to zero.

And so − let me try that again, I'm sorry. Let's say I want to ask − I want to look at the [inaudible] element of the vector TY, and I want to know is this one or zero. All right. Well, this will be one. The [inaudible] element of the vector TY will be equal to one if, and only if Y is equal to I.

Because for example, if Y is equal to one, then only the first element of this vector will be zero. If Y is equal to two, then only the second element of the vector will be zero and so on. So the question of whether or not − whether the [inaudible] element of this vector, TY, is equal to one is answered by just asking is Y equal to I.

Okay. If you're still not quite sure why that's true, go home and think about it a bit more. And I think I − and take a look at the lecture notes as well, maybe that'll help. At least for now, only just take my word for it.

So let's go ahead and write out the distribution for the [inaudible] in an exponential family form. So PFY is equal to phi one. Indicator Y equals one times phi two. Indicator Y equals to up to phi K times indicator Y equals K. And again, phi K is not a parameter of the distribution. Phi K is a shorthand for one minus phi one minus phi two minus the rest.

And so using this equation on the left as well, I can also write this as phi one times TY one, phi two, TY two, dot, dot, dot. Phi K minus one, TY, K minus one times phi K. And then one minus [inaudible]. That should be K.

And it turns out − it takes some of the steps of algebra that I don't have time to show. It turns out, you can simplify this into − well, the exponential family form where [inaudible] is a vector, this is a K minus one-dimensional vector, and − well, okay.

So deriving this is a few steps of algebra that you can work out yourself, but I won't do here. And so using my definition for TY, and by choosing [inaudible] A and B this way, I can take my distribution from [inaudible] and write it out in a form of an exponential family distribution.

It turns out also that − let's see. [Inaudible], right. One of the things we did was we also had [inaudible] as a function of phi, and then we inverted that to write out phi as a function of [inaudible]. So it turns out you can do that as well.

So this defines [inaudible] as a function of the [inaudible] distributions parameters phi. So you can take

this relationship between [inaudible] and phi and invert it, and write out phi as a function of [inaudible]. And it turns out, you get that phi I is equal to [inaudible] – excuse me. And you get that phi I is equal to [inaudible] I of one plus that.

And the way you do this is you just – this defines [inaudible] as a function of the phi, so if you take this and solve for [inaudible], you end up with this. And this is – again, there are a couple of steps of algebra that I'm just not showing.

And then lastly, using our assumption that the [inaudible] are a linear function of the [inaudible] axis, phi I is therefore equal to E to the theta I, transpose X, divided by one plus sum over J equals one, to K minus one, E to the theta J, transpose X. And this is just using the fact that [inaudible] I equals theta I, transpose X, which was our earlier design choice from generalized linear models.

So we're just about down. So my learning algorithm [inaudible]. I'm going to think of it as [inaudible] the expected value of TY given X and [inaudible] by theta. So TY was this vector indicator function. So T one was indicator Y equals one, down to indicator Y equals K minus one. All right. So I want my learning algorithm to output this; the expected value of this vector of indicator functions.

The expected value of indicator Y equals one is just the probability that Y equals one, which is given by phi one. So I have a random variable that's one whenever Y is equal to one and zero otherwise, so the expected value of that, of this indicator Y equals one is just the probability that Y equals one, which is given by phi one.

And therefore, by what we were taught earlier, this is therefore [inaudible] to the theta one, transpose X over – well – okay. And so my learning algorithm will output the probability that Y equals one, Y equals two, up to Y equals K minus one. And these probabilities are going to be parameterized by these functions like these.

And so just to give this algorithm a name, this algorithm is called softmax regression, and is widely thought of as the generalization of logistic regression, which is regression of two classes. Is widely thought of as a generalization of logistic regression to the case of K classes rather than two classes.

And so just to be very concrete about what you do, right. So you have a machine-learning problem, and you want to apply softmax regression to it. So generally, work for the entire derivation [inaudible]. I think the question you had is about how to fit parameters.

So let's say you have a machine-learning problem, and Y takes on one of K classes. What you do is you sit down and say, "Okay, I wanna model Y as being [inaudible] given any X and then theta." And so you chose [inaudible] as the exponential family. Then you sort of turn the crank. And everything else I wrote down follows automatically from you have made the choice of using [inaudible] distribution as your choice of exponential family.

And then what you do is you then have this training set, X, I, Y, I up to X, M, Y, M. So you're doing the training set. We're now [inaudible] the value of Y takes on one of K possible values.

And what you do is you then find the parameters of the model by maximum likelihood. So you write down the likelihood of the parameters, and you maximize the likelihood.

So what's the likelihood? Well, the likelihood, as usual, is the product of your training set of P of YI given XI parameterized by theta. That's the likelihood, same as we had before. And that's product of your training set of – let me write these down now. YI equals one times phi two of indicator YI equals two, dot, dot, dot, to phi K of indicator YI equals K.

Where, for example, phi one depends on theta through this formula. It is E to the theta one, transpose X over one plus sum over J – well, that formula I had just now. And so phi one here is really a shorthand for this formula, and similarly for phi two and so on, up to phi K, where phi K is one minus all of these things. All right.

So this is a –this formula looks more complicated than it really is. What you really do is you write this down, then you take logs, compute a derivative of this formula [inaudible] theta, and apply say gradient ascent to maximize the likelihood.

Student: What are the rows of theta? [Inaudible] it's just been a vector, right? And now it looks like it's two-dimensional.

Instructor (Andrew Ng): Yeah. In the notation of the [inaudible] I think have theta one through theta

K minus one. I've been thinking of each of these as − and N plus one-dimensional vector. If X is N plus one-dimensional, then I've been − see, I think if you have a set of parameters comprising K minus one vectors, and each of these is a − you could group all of these together and make these, but I just haven't been doing that. [Inaudible] the derivative of K minus one parameter vectors.

Student:[Inaudible], what do they correspond to?

Instructor (Andrew Ng):[Inaudible]. We're sort of out of time. Let me take that offline. It's hard to answer in the same way that the logistic regression − what does theta correspond to in logistic regression? You can sort of answer that as sort of −

Student: Yeah. It's kind of like the [inaudible] feature −

Instructor (Andrew Ng): Yeah. Sort of similar interpretation, yeah. That's good. I think I'm running a little bit late. Why don't I − why don't we officially close for the day, but you can come up if you more questions and take them offline. Thanks.

[End of Audio]

Duration: 76 minutes

1.5 Machine Learning Lecture 5[5]

Figure 1.5

MachineLearning-Lecture05

Instructor (Andrew Ng): Okay, good morning. Just one quick announcement and reminder, the project guidelines handout was posted on the course website last week. So if you haven't yet downloaded it and looked at it, please do so. It just contains the guidelines for the project proposal and the project milestone, and the final project presentation.

So what I want to do today is talk about a different type of learning algorithm, and, in particular, start to talk about generative learning algorithms and the specific algorithm called Gaussian Discriminant Analysis. Take a slight digression, talk about Gaussians, and I'll briefly discuss generative versus discriminative learning algorithms, and then hopefully wrap up today's lecture with a discussion of Naive Bayes and the Laplace Smoothing.

So just to motivate our discussion on generative learning algorithms, right, so by way of contrast, the source of classification algorithms we've been talking about I think of algorithms that do this. So you're given a training set, and if you run an algorithm right, we just see progression on those training sets.

The way I think of logistic regression is that it's trying to find − look at the date and is trying to find a straight line to divide the crosses and O's, right? So it's, sort of, trying to find a straight line. Let me − just make the days a bit noisier. Trying to find a straight line that separates out the positive and the negative classes as well as pass the law, right?

And, in fact, it shows it on the laptop. Maybe just use the screens or the small monitors for this. In fact, you can see there's the data set with logistic regression, and so I've initialized the parameters randomly, and so logistic regression is, kind of, the outputting − it's the, kind of, hypothesis that iteration zero is that straight line shown in the bottom right.

And so after one iteration and creating descent, the straight line changes a bit. After two iterations, three, four, until logistic regression converges and has found the straight line that, more or less, separates the positive and negative class, okay? So you can think of this as logistic regression, sort of, searching for a line that separates the positive and the negative classes.

---

[5]This content is available online at <http://cnx.org/content/m45890/1.2/>.

What I want to do today is talk about an algorithm that does something slightly different, and to motivate us, let's use our old example of trying to classify the team malignant cancer and benign cancer, right? So a patient comes in and they have a cancer, you want to know if it's a malignant or a harmful cancer, or if it's a benign, meaning a harmless cancer.

So rather than trying to find the straight line to separate the two classes, here's something else we could do. We can go from our training set and look at all the cases of malignant cancers, go through, you know, look for our training set for all the positive examples of malignant cancers, and we can then build a model for what malignant cancer looks like. Then we'll go for our training set again and take out all of the examples of benign cancers, and then we'll build a model for what benign cancers look like, okay?

And then when you need to classify a new example, when you have a new patient, and you want to decide is this cancer malignant or benign, you then take your new cancer, and you match it to your model of malignant cancers, and you match it to your model of benign cancers, and you see which model it matches better, and depending on which model it matches better to, you then predict whether the new cancer is malignant or benign, okay?

So what I just described, just this cross of methods where you build a second model for malignant cancers and a separate model for benign cancers is called a generative learning algorithm, and let me just, kind of, formalize this. So in the models that we've been talking about previously, those were actually all discriminative learning algorithms, and studied more formally, a discriminative learning algorithm is one that either learns PFY given X directly, or even learns a hypothesis that outputs value 0, 1 directly, okay? So logistic regression is an example of a discriminative learning algorithm.

In contrast, a generative learning algorithm of models PFX given Y. The probability of the features given the class label, and as a technical detail, it also models PFY, but that's a less important thing, and the interpretation of this is that a generative model builds a probabilistic model for what the features looks like, conditioned on the class label, okay? In other words, conditioned on whether a cancer is malignant or benign, it models probability distribution over what the features of the cancer looks like.

Then having built this model – having built a model for PFX given Y and PFY, then by Bayes rule, obviously, you can compute PFY given 1, conditioned on X. This is just PFX given Y = 1 × PFX ÷ PFX, and, if necessary, you can calculate the denominator using this, right? And so by modeling PFX given Y and modeling PFY, you can actually use Bayes rule to get back to PFY given X, but a generative model – generative learning algorithm starts in modeling PFX given Y, rather than PFY given X, okay?

We'll talk about some of the tradeoffs, and why this may be a better or worse idea than a discriminative model a bit later. Let's go for a specific example of a generative learning algorithm, and for this specific motivating example, I'm going to assume that your input feature is X and RN and are continuous values, okay?

And under this assumption, let me describe to you a specific algorithm called Gaussian Discriminant Analysis, and the, I guess, core assumption is that we're going to assume in the Gaussian discriminant analysis model of that PFX given Y is Gaussian, okay?

So actually just raise your hand, how many of you have seen a multivariate Gaussian before – not a 1D Gaussian, but the higher range though? Okay, cool, like maybe half of you, two-thirds of you. So let me just say a few words about Gaussians, and for those of you that have seen it before, it'll be a refresher.

So we say that a random variable Z is distributed Gaussian, multivariate Gaussian as – and the script N for normal with parameters mean U and covariance sigma squared. If Z has a density 1 over 2 Pi, sigma 2, okay? That's the formula for the density as a generalization of the one dimension of Gaussians and no more the familiar bell-shape curve. It's a high dimension vector value random variable Z.

Don't worry too much about this formula for the density. You rarely end up needing to use it, but the two key quantities are this vector mew is the mean of the Gaussian and this matrix sigma is the covariance matrix – covariance, and so sigma will be equal to, right, the definition of covariance of a vector valued random variable is X - U, X - V conspose, okay?

And, actually, if this doesn't look familiar to you, you might re-watch the discussion section that the TAs held last Friday or the one that they'll be holding later this week on, sort of, a recap of probability, okay?

So multi-grade Gaussians is parameterized by a mean and a covariance, and let me just – can I have

the laptop displayed, please? I'll just go ahead and actually show you, you know, graphically, the effects of varying a Gaussian – varying the parameters of a Gaussian. So what I have up here is the density of a zero mean Gaussian with covariance matrix equals the identity. The covariance matrix is shown in the upper right-hand corner of the slide, and there's the familiar bell-shaped curve in two dimensions.

And so if I shrink the covariance matrix, instead of covariance your identity, if I shrink the covariance matrix, then the Gaussian becomes more peaked, and if I widen the covariance, so like same = 2, 2, then the distribution – well, the density becomes more spread out, okay?

Those vectors stand at normal, identity covariance one. If I increase the diagonals of a covariance matrix, right, if I make the variables correlated, and the Gaussian becomes flattened out in this X = Y direction, and increase it even further, then my variables, X and Y, right – excuse me, it goes Z1 and Z2 are my two variables on a horizontal axis become even more correlated.

I'll just show the same thing in contours. The standard normal of distribution has contours that are – they're actually circles. Because of the aspect ratio, these look like ellipses. These should actually be circles, and if you increase the off diagonals of the Gaussian covariance matrix, then it becomes ellipses aligned along the, sort of, 45 degree angle in this example.

This is the same thing. Here's an example of a Gaussian density with negative covariances. So now the correlation goes the other way, so that even strong [inaudible] of covariance and the same thing in contours. This is a Gaussian with negative entries on the diagonals and even larger entries on the diagonals, okay?

And other parameter for the Gaussian is the mean parameters, so if this is – with mew0, and as he changed the mean parameter, this is mew equals 0.15, the location of the Gaussian just moves around, okay?

All right. So that was a quick primer on what Gaussians look like, and here's as a roadmap or as a picture to keep in mind, when we described the Gaussian discriminant analysis algorithm, this is what we're going to do. Here's the training set, and in the Gaussian discriminant analysis algorithm, what I'm going to do is I'm going to look at the positive examples, say the crosses, and just looking at only the positive examples, I'm gonna fit a Gaussian distribution to the positive examples, and so maybe I end up with a Gaussian distribution like that, okay? So there's PFX given Y = 1.

And then I'll look at the negative examples, the O's in this figure, and I'll fit a Gaussian to that, and maybe I get a Gaussian centered over there. This is the concept of my second Gaussian, and together – we'll say how later – together these two Gaussian densities will define a separator for these two classes, okay?

And it'll turn out that the separator will turn out to be a little bit different from what logistic regression gives you. If you run logistic regression, you actually get the division bound to be shown in the green line, whereas Gaussian discriminant analysis gives you the blue line, okay?

Switch back to chalkboard, please. All right. Here's the Gaussian discriminant analysis model, put into model PFY as a Bernoulli random variable as usual, but as a Bernoulli random variable and parameterized by parameter phi; you've seen this before. Model PFX given Y = 0 as a Gaussian – oh, you know what?

Yeah, yes, excuse me. I thought this looked strange. This should be a sigma, determined in a sigma to the one-half of the denominator there. It's no big deal. It was – yeah, well, okay. Right. I was listing the sigma to the determining the sigma to the one-half on a previous board, excuse me.

Okay, and so I model PFX given Y = 0 as a Gaussian with mean mew0 and covariance sigma to the sigma to the minus one-half, and – okay? And so the parameters of this model are phi, mew0, mew1, and sigma, and so I can now write down the likelihood of the parameters as – oh, excuse me, actually, the log likelihood of the parameters as the log of that, right?

So, in other words, if I'm given the training set, then they can write down the log likelihood of the parameters as the log of, you know, the probative probabilities of PFXI, YI, right? And this is just equal to that where each of these terms, PFXI given YI, or PFYI is then given by one of these three equations on top, okay?

And I just want to contrast this again with discriminative learning algorithms, right? So to give this a name, I guess, this sometimes is actually called the Joint Data Likelihood – the Joint Likelihood, and let me just contrast this with what we had previously when we're talking about logistic regression. Where I said with the log likelihood of the parameter's theater was log of a product I = 1 to M, PFYI given XI and parameterized by a theater, right?

So back where we're fitting logistic regression models or generalized learning models, we're always modeling PFYI given XI and parameterized by a theater, and that was the conditional likelihood, okay, in which we're modeling PFYI given XI, whereas, now, regenerative learning algorithms, we're going to look at the joint likelihood which is PFXI, YI, okay?

So let's see. So given the training sets and using the Gaussian discriminant analysis model to fit the parameters of the model, we'll do maximize likelihood estimation as usual, and so you maximize your L with respect to the parameters phi, mew0, mew1, sigma, and so if we find the maximum likelihood estimate of parameters, you find that phi is – the maximum likelihood estimate is actually no surprise, and I'm writing this down mainly as a practice for indicating notation, all right?

So the maximum likelihood estimate for phi would be Sum over I, YI ÷ M, or written alternatively as Sum over – all your training examples of indicator YI = 1 ÷ M, okay? In other words, maximum likelihood estimate for a newly parameter phi is just the faction of training examples with label one, with Y equals 1. Maximum likelihood estimate for mew0 is this, okay? You should stare at this for a second and see if it makes sense.

Actually, I'll just write on the next one for mew1 while you do that. Okay? So what this is is what the denominator is sum of your training sets indicated YI = 0. So for every training example for which YI = 0, this will increment the count by one, all right?

So the denominator is just the number of examples with label zero, all right? And then the numerator will be, let's see, Sum from I = 1 for M, or every time YI is equal to 0, this will be a one, and otherwise, this thing will be zero, and so this indicator function means that you're including only the times for which YI is equal to one – only the turns which Y is equal to zero because for all the times where YI is equal to one, this sum and will be equal to zero, and then you multiply that by XI, and so the numerator is really the sum of XI's corresponding to examples where the class labels were zero, okay? Raise your hand if this makes sense. Okay, cool.

So just to say this fancifully, this just means look for your training set, find all the examples for which Y = 0, and take the average of the value of X for all your examples which Y = 0. So take all your negative fitting examples and average the values for X and that's mew0, okay?

If this notation is still a little bit cryptic – if you're still not sure why this equation translates into what I just said, do go home and stare at it for a while until it just makes sense. This is, sort of, no surprise. It just says to estimate the mean for the negative examples, take all your negative examples, and average them. So no surprise, but this is a useful practice to indicate a notation.

[Inaudible] divide the maximum likelihood estimate for sigma. I won't do that. You can read that in the notes yourself. And so having fit the parameters find mew0, mew1, and sigma to your data, well, you now need to make a prediction. You know, when you're given a new value of X, when you're given a new cancer, you need to predict whether it's malignant or benign.

Your prediction is then going to be, let's say, the most likely value of Y given X. I should write semicolon the parameters there. I'll just give that – which is the [inaudible] of a Y by Bayes rule, all right? And that is, in turn, just that because the denominator PFX doesn't depend on Y, and if PFY is uniform.

In other words, if each of your constants is equally likely, so if PFY takes the same value for all values of Y, then this is just arc X over Y, PFX given Y, okay?

This happens sometimes, maybe not very often, so usually you end up using this formula where you compute PFX given Y and PFY using your model, okay?

Student: Can you give us arc x?

Instructor (Andrew Ng): Oh, let's see. So if you take – actually let me. So the min of – arcomatics means the value for Y that maximizes this.

Student: Oh, okay.

Instructor (Andrew Ng): So just for an example, the min of X - 5 squared is 0 because by choosing X equals 5, you can get this to be zero, and the argument over X of X - 5 squared is equal to 5 because 5 is the value of X that makes this minimize, okay? Cool. Thanks for asking that.

Instructor (Andrew Ng): Okay. Actually any other questions about this? Yeah?

Student: Why is distributive removing? Why isn't [inaudible] –

Instructor (Andrew Ng):Oh, I see. By uniform I meant – I was being loose here. I meant if PFY = 0 is equal to PFY = 1, or if Y is the uniform distribution over the set 0 and 1.

Student:Oh.

Instructor (Andrew Ng):I just meant – yeah, if PFY = 0 zero = PFY given 1. That's all I mean, see? Anything else?

All right. Okay. So it turns out Gaussian discriminant analysis has an interesting relationship to logistic regression. Let me illustrate that. So let's say you have a training set – actually let me just go ahead and draw 1D training set, and that will kind of work, yes, okay.

So let's say we have a training set comprising a few negative and a few positive examples, and let's say I run Gaussian discriminate analysis. So I'll fit Gaussians to each of these two densities – a Gaussian density to each of these two – to my positive and negative training examples, and so maybe my positive examples, the X's, are fit with a Gaussian like this, and my negative examples I will fit, and you have a Gaussian that looks like that, okay?

Now, I hope this [inaudible]. Now, let's vary along the X axis, and what I want to do is I'll overlay on top of this plot. I'm going to plot PFY = 1 – no, actually, given X for a variety of values X, okay? So I actually realize what I should have done. I'm gonna call the X's the negative examples, and I'm gonna call the O's the positive examples. It just makes this part come in better.

So let's take a value of X that's fairly small. Let's say X is this value here on a horizontal axis. Then what's the probability of Y being equal to one conditioned on X? Well, the way you calculate that is you write PFY = 1 given X, and then you plug in all these formulas as usual, right? It's PFX given Y = 1, which is your Gaussian density, times PFY = 1, you know, which is essentially – this is just going to be equal to phi, and then divided by, right, PFX, and then this shows you how you can calculate this.

By using these two Gaussians and my phi on PFY, I actually compute what PFY = 1 given X is, and in this case, if X is this small, clearly it belongs to the left Gaussian. It's very unlikely to belong to a positive class, and so it'll be very small; it'll be very close to zero say, okay? And then we can increment the value of X a bit, and study a different value of X, and plot what is the PFY given X – PFY = 1 given X, and, again, it'll be pretty small.

Let's use a point like that, right? At this point, the two Gaussian densities have equal value, and if I ask if X is this value, right, shown by the arrow, what's the probably of Y being equal to one for that value of X? Well, you really can't tell, so maybe it's about 0.5, okay?

And if you fill in a bunch more points, you get a curve like that, and then you can keep going. Let's say for a point like that, you can ask what's the probability of X being one? Well, if it's that far out, then clearly, it belongs to this rightmost Gaussian, and so the probability of Y being a one would be very high; it would be almost one, okay?

And so you can repeat this exercise for a bunch of points. All right, compute PFY equals one given X for a bunch of points, and if you connect up these points, you find that the curve you get [Pause] plotted takes a form of sigmoid function, okay?

So, in other words, when you make the assumptions under the Gaussian discriminant analysis model, that PFX given Y is Gaussian, when you go back and compute what PFY given X is, you actually get back exactly the same sigmoid function that we're using which is the progression, okay?

But it turns out the key difference is that Gaussian discriminant analysis will end up choosing a different position and a steepness of the sigmoid than would logistic regression. Is there a question?

Student:I'm just wondering, the Gaussian of PFY [inaudible] you do?

Instructor (Andrew Ng):No, let's see. The Gaussian – so this Gaussian is PFX given Y = 1, and this Gaussian is PFX given Y = 0; does that make sense? Anything else?

Student:Okay.

Instructor (Andrew Ng):Yeah?

Student:When you drawing all the dots, how did you decide what Y given PFX was?

Instructor (Andrew Ng):What – say that again.

Student:I'm sorry. Could you go over how you figured out where to draw each dot?

Instructor (Andrew Ng): Let's see, okay. So the computation is as follows, right? The steps are I have the training sets, and so given my training set, I'm going to fit a Gaussian discriminant analysis model to it, and what that means is I'll build a model for PFX given Y = 1. I'll build a model for PFX given Y = 0, and I'll also fit a Bernoulli distribution to PFY, okay?

So, in other words, given my training set, I'll fit PFX given Y and PFY to my data, and now I've chosen my parameters of find mew0, mew1, and the sigma, okay? Then this is the process I went through to plot all these dots, right? It's just I pick a point in the X axis, and then I compute PFY given X for that value of X, and PFY given 1 conditioned on X will be some value between zero and one. It'll be some real number, and whatever that real number is, I then plot it on the vertical axis, okay?

And the way I compute PFY = 1 conditioned on X is I would use these quantities. I would use PFX given Y and PFY, and, sort of, plug them into Bayes rule, and that allows me to compute PFY given X from these three quantities; does that make sense?

Student: Yeah.

Instructor (Andrew Ng): Was there something more that –

Student: And how did you model PFX; is that –

Instructor (Andrew Ng): Oh, okay. Yeah, so – well, got this right here. So PFX can be written as, right, so PFX given Y = 0 × PFY = 0 + PFX given Y = 1, PFY = 1, right? And so each of these terms, PFX given Y and PFY, these are terms I can get out of, directly, from my Gaussian discriminant analysis model. Each of these terms is something that my model gives me directly, so plugged in as the denominator, and by doing that, that's how I compute PFY = 1 given X, make sense?

Student: Thank you.

Instructor (Andrew Ng): Okay. Cool. So let's talk a little bit about the advantages and disadvantages of using a generative learning algorithm, okay? So in the particular case of Gaussian discriminant analysis, we assume that X conditions on Y is Gaussian, and the argument I showed on the previous chalkboard, I didn't prove it formally, but you can actually go back and prove it yourself is that if you assume X given Y is Gaussian, then that implies that when you plot Y given X, you find that – well, let me just write logistic posterior, okay?

And the argument I showed just now, which I didn't prove; you can go home and prove it yourself, is that if you assume X given Y is Gaussian, then that implies that the posterior distribution or the form of PFY = 1 given X is going to be a logistic function, and it turns out this implication in the opposite direction does not hold true, okay?

In particular, it actually turns out – this is actually, kind of, cool. It turns out that if you're seeing that X given Y = 1 is Hessian with parameter lambda 1, and X given Y = 0, is Hessian with parameter lambda 0. It turns out if you assumed this, then that also implies that PFY given X is logistic, okay?

So there are lots of assumptions on X given Y that will lead to PFY given X being logistic, and, therefore, this, the assumption that X given Y being Gaussian is the stronger assumption than the assumption that Y given X is logistic, okay? Because this implies this, right? That means that this is a stronger assumption than this because this, the logistic posterior holds whenever X given Y is Gaussian but not vice versa.

And so this leaves some of the tradeoffs between Gaussian discriminant analysis and logistic regression, right? Gaussian discriminant analysis makes a much stronger assumption that X given Y is Gaussian, and so when this assumption is true, when this assumption approximately holds, if you plot the data, and if X given Y is, indeed, approximately Gaussian, then if you make this assumption, explicit to the algorithm, then the algorithm will do better because it's as if the algorithm is making use of more information about the data. The algorithm knows that the data is Gaussian, right? And so if the Gaussian assumption, you know, holds or roughly holds, then Gaussian discriminant analysis may do better than logistic regression.

If, conversely, if you're actually not sure what X given Y is, then logistic regression, the discriminant algorithm may do better, and, in particular, use logistic regression, and maybe you see [inaudible] before the data was Gaussian, but it turns out the data was actually Poisson, right? Then logistic regression will still do perfectly fine because if the data were actually Poisson, then PFY = 1 given X will be logistic, and it'll do perfectly fine, but if you assumed it was Gaussian, then the algorithm may go off and do something that's not as good, okay?

So it turns out that − right. So it's slightly different. It turns out the real advantage of generative learning algorithms is often that it requires less data, and, in particular, data is never really exactly Gaussian, right? Because data is often approximately Gaussian; it's never exactly Gaussian.

And it turns out, generative learning algorithms often do surprisingly well even when these modeling assumptions are not met, but one other tradeoff is that by making stronger assumptions about the data, Gaussian discriminant analysis often needs less data in order to fit, like, an okay model, even if there's less training data.

Whereas, in contrast, logistic regression by making less assumption is more robust to your modeling assumptions because you're making a weaker assumption; you're making less assumptions, but sometimes it takes a slightly larger training set to fit than Gaussian discriminant analysis. Question?

Student: In order to meet any assumption about the number [inaudible], plus here we assume that PFY = 1, equal two number of. [Inaudible]. Is true when the number of samples is marginal?

Instructor (Andrew Ng):Okay. So let's see. So there's a question of is this true − what was that? Let me translate that differently. So the marving assumptions are made independently of the size of your training set, right? So, like, in least/great regression − well, in all of these models I'm assuming that these are random variables flowing from some distribution, and then, finally, I'm giving a single training set and that as for the parameters of the distribution, right?

Student: So what's the probability of Y = 1?

Instructor (Andrew Ng):Probability of Y + 1?

Student: Yeah, you used the −

Instructor (Andrew Ng):Sort of, this like − back to the philosophy of mass molecular estimation, right? I'm assuming that they're PFY is equal to phi to the Y, Y - phi to the Y or Y - Y. So I'm assuming that there's some true value of Y generating all my data, and then − well, when I write this, I guess, maybe what I should write isn't − so when I write this, I guess there are already two values of phi. One is there's a true underlying value of phi that guards the use to generate the data, and then there's the maximum likelihood estimate of the value of phi, and so when I was writing those formulas earlier, those formulas are writing for phi, and mew0, and mew1 were really the maximum likelihood estimates for phi, mew0, and mew1, and that's different from the true underlying values of phi, mew0, and mew1, but −

Student:[Off mic].

Instructor (Andrew Ng):Yeah, right. So maximum likelihood estimate comes from the data, and there's some, sort of, true underlying value of phi that I'm trying to estimate, and my maximum likelihood estimate is my attempt to estimate the true value, but, you know, by notational and convention often are just right as that as well without bothering to distinguish between the maximum likelihood value and the true underlying value that I'm assuming is out there, and that I'm only hoping to estimate.

Actually, yeah, so for the sample of questions like these about maximum likelihood and so on, I hope to tease to the Friday discussion section as a good time to ask questions about, sort of, probabilistic definitions like these as well. Are there any other questions? No, great. Okay.

So, great. Oh, it turns out, just to mention one more thing that's, kind of, cool. I said that if X given Y is Poisson, and you also go logistic posterior, it actually turns out there's a more general version of this. If you assume X given Y = 1 is exponential family with parameter A to 1, and then you assume X given Y = 0 is exponential family with parameter A to 0, then this implies that PFY = 1 given X is also logistic, okay? And that's, kind of, cool. It means that Y given X could be − I don't know, some strange thing. It could be gamma because we've seen Gaussian right next to the − I don't know, gamma exponential. They're actually a beta.

I'm just rattling off my mental list of exponential family extrusions.

It could be any one of those things, so [inaudible] the same exponential family distribution for the two classes with different natural parameters than the posterior PFY given 1 given X − PFY = 1 given X would be logistic, and so this shows the robustness of logistic regression to the choice of modeling assumptions because it could be that the data was actually, you know, gamma distributed, and just still turns out to be logistic. So it's the robustness of logistic regression to modeling assumptions.

And this is the density. I think, early on I promised two justifications for where I pulled the logistic

function out of the hat, right? So one was the exponential family derivation we went through last time, and this is, sort of, the second one. That all of these modeling assumptions also lead to the logistic function. Yeah?

Student:[Off mic].

Instructor (Andrew Ng): Oh, that Y = 1 given as the logistic then this implies that, no. This is also not true, right? Yeah, so this exponential family distribution implies Y = 1 is logistic, but the reverse assumption is also not true. There are actually all sorts of really bizarre distributions for X that would give rise to logistic function, okay?

Okay. So let's talk about – those are first generative learning algorithm. Maybe I'll talk about the second generative learning algorithm, and the motivating example, actually this is called a Naive Bayes algorithm, and the motivating example that I'm gonna use will be spam classification.

All right. So let's say that you want to build a spam classifier to take your incoming stream of email and decide if it's spam or not. So let's see. Y will be 0 or 1, with 1 being spam email and 0 being non-spam, and the first decision we need to make is, given a piece of email, how do you represent a piece of email using a feature vector X, right? So email is just a piece of text, right? Email is like a list of words or a list of ASCII characters.

So I can represent email as a feature of vector X. So we'll use a couple of different representations, but the one I'll use today is we will construct the vector X as follows. I'm gonna go through my dictionary, and, sort of, make a listing of all the words in my dictionary, okay?

So the first word is RA. The second word in my dictionary is Aardvark, ausworth, okay? You know, and somewhere along the way you see the word "buy" in the spam email telling you to buy stuff. Tell you how you collect your list of words, you know, you won't find CS229, right, course number in a dictionary, but if you collect a list of words via other emails you've gotten, you have this list somewhere as well, and then the last word in my dictionary was zicmergue, which pertains to the technological chemistry that deals with the fermentation process in brewing.

So say I get a piece of email, and what I'll do is I'll then scan through this list of words, and wherever a certain word appears in my email, I'll put a 1 there. So if a particular email has the word "aid" then that's 1. You know, my email doesn't have the words ausworth or aardvark, so it gets zeros. And again, a piece of email, they want me to buy something, CS229 doesn't occur, and so on, okay? So this would be one way of creating a feature vector to represent a piece of email.

Now, let's throw the generative model out for this. Actually, let's use this. In other words, I want to model PFX given Y. The given Y = 0 or Y = 1, all right? And my feature vectors are going to be 0, 1 to the N. It's going to be these split vectors, binary value vectors. They're N dimensional. Where N may be on the order of, say, 50,000, if you have 50,000 words in your dictionary, which is not atypical. So values from – I don't know, mid-thousands to tens of thousands is very typical for problems like these.

And, therefore, there two to the 50,000 possible values for X, right? So two to 50,000 possible bit vectors of length 50,000, and so one way to model this is the multinomial distribution, but because there are two to the 50,000 possible values for X, I would need two to the 50,000, but maybe -1 parameters, right? Because you have this sum to 1, right? So -1. And this is clearly way too many parameters to model using the multinomial distribution over all two to 50,000 possibilities.

So in a Naive Bayes algorithm, we're going to make a very strong assumption on PFX given Y, and, in particular, I'm going to assume – let me just say what it's called; then I'll write out what it means. I'm going to assume that the XI's are conditionally independent given Y, okay?

Let me say what this means. So I have that PFX1, X2, up to X50,000, right, given the Y. By the key rule of probability, this is PFX1 given Y times PFX2 given Y, X1 times PF – I'll just put dot, dot, dot. I'll just write 1, 1 × dot, dot, dot up to, you know, well – whatever. You get the idea, up to PFX50,000, okay?

So this is the chain were of probability. This always holds. I've not made any assumption yet, and now, we're gonna meet what's called the Naive Bayes assumption, or this assumption that X defies a conditionally independent given Y. Going to assume that – well, nothing changes for the first term, but I'm gonna assume that PFX3 given Y, X1 is equal to PFX2 given the Y. I'm gonna assume that that term's equal to PFX3 given the Y, and so on, up to PFX50,000 given Y, okay? Or just written more compactly, means assume

that PFX1, PFX50,000 given Y is the product from I = 1 to 50,000 or PFXI given the Y, okay?

And stating informally what this means is that I'm, sort of, assuming that – so unless you know the cost label Y, so long as you know whether this is spam or not spam, then knowing whether the word "A" appears in email does not affect the probability of whether the word "Ausworth" appears in the email, all right?

And, in other words, there's assuming – once you know whether an email is spam or not spam, then knowing whether other words appear in the email won't help you predict whether any other word appears in the email, okay? And, obviously, this assumption is false, right? This assumption can't possibly be true. I mean, if you see the word – I don't know, CS229 in an email, you're much more likely to see my name in the email, or the TA's names, or whatever.

So this assumption is normally just false under English, right, for normal written English, but it turns out that despite this assumption being, sort of, false in the literal sense, the Naive Bayes algorithm is, sort of, an extremely effective algorithm for classifying text documents into spam or not spam, for classifying your emails into different emails for your automatic view, for looking at web pages and classifying whether this webpage is trying to sell something or whatever. It turns out, this assumption works very well for classifying text documents and for other applications too that I'll talk a bit about later.

As a digression that'll make sense only to some of you. Let me just say that if you're familiar with Bayesian X world, say graphical models, the Bayesian network associated with this model looks like this, and you're assuming that this is random variable Y that then generates X1, X2, through X50,000, okay? If you've not seen the Bayes Net before, if you don't know your graphical model, just ignore this. It's not important to our purposes, but if you've seen it before, that's what it will look like.

Okay. So the parameters of the model are as follows with phi FI given Y = 1, which is probably FX = 1 or XI = 1 given Y = 1, phi I given Y = 0, and phi Y, okay? So these are the parameters of the model, and, therefore, to fit the parameters of the model, you can write down the joint likelihood, right, is equal to, as usual, okay?

So given the training sets, you can write down the joint likelihood of the parameters, and then when you do maximum likelihood estimation, you find that the maximum likelihood estimate of the parameters are – they're really, pretty much, what you'd expect. Maximum likelihood estimate for phi J given Y = 1 is sum from I = 1 to M, indicator XIJ = 1, YI = 1, okay?

And this is just a, I guess, stated more simply, the numerator just says, "Run for your entire training set, some [inaudible] examples, and count up the number of times you saw word "Jay" in a piece of email for which the label Y was equal to 1." So, in other words, look through all your spam emails and count the number of emails in which the word "Jay" appeared out of all your spam emails, and the denominator is, you know, sum from I = 1 to M, the number of spam. The denominator is just the number of spam emails you got.

And so this ratio is in all your spam emails in your training set, what fraction of these emails did the word "Jay" appear in – did the, "Jay" you wrote in your dictionary appear in? And that's the maximum likelihood estimate for the probability of seeing the word "Jay" conditions on the piece of email being spam, okay? And similar to your maximum likelihood estimate for phi Y is pretty much what you'd expect, right? Okay?

And so having estimated all these parameters, when you're given a new piece of email that you want to classify, you can then compute PFY given X using Bayes rule, right? Same as before because together these parameters gives you a model for PFX given Y and for PFY, and by using Bayes rule, given these two terms, you can compute PFX given Y, and there's your spam classifier, okay? Turns out we need one more elaboration to this idea, but let me check if there are questions about this so far.

Student: So does this model depend on the number of inputs?

Instructor (Andrew Ng):What do you mean, number of inputs, the number of features?

Student: No, number of samples.

Instructor (Andrew Ng):Well, N is the number of training examples, so this given M training examples, this is the formula for the maximum likelihood estimate of the parameters, right? So other questions, does it make sense? Or M is the number of training examples, so when you have M training examples, you plug them into this formula, and that's how you compute the maximum likelihood estimates.

Student: Is training examples you mean M is the number of emails?

Instructor (Andrew Ng): Yeah, right. So, right. So it's, kind of, your training set. I would go through all the email I've gotten in the last two months and label them as spam or not spam, and so you have – I don't know, like, a few hundred emails labeled as spam or not spam, and that will comprise your training sets for X1 and Y1 through XM, YM, where X is one of those vectors representing which words appeared in the email and Y is 0, 1 depending on whether they equal spam or not spam, okay?

Student: So you are saying that this model depends on the number of examples, but the last model doesn't depend on the models, but your phi is the same for either one.

Instructor (Andrew Ng): They're different things, right? There's the model which is – the modeling assumptions aren't made very well. I'm assuming that – I'm making the Naive Bayes assumption. So the probabilistic model is an assumption on the joint distribution of X and Y. That's what the model is, and then I'm given a fixed number of training examples. I'm given M training examples, and then it's, like, after I'm given the training sets, I'll then go in to write the maximum likelihood estimate of the parameters, right? So that's, sort of, maybe we should take that offline for – yeah, ask a question?

Student: Then how would you do this, like, if this [inaudible] didn't work?

Instructor (Andrew Ng): Say that again.

Student: How would you do it, say, like the 50,000 words –

Instructor (Andrew Ng): Oh, okay. How to do this with the 50,000 words, yeah. So it turns out this is, sort of, a very practical question, really. How do I count this list of words? One common way to do this is to actually find some way to count a list of words, like go through all your emails, go through all the – in practice, one common way to count a list of words is to just take all the words that appear in your training set.

That's one fairly common way to do it, or if that turns out to be too many words, you can take all words that appear at least three times in your training set. So words that you didn't even see three times in the emails you got in the last two months, you discard. So those are – I was talking about going through a dictionary, which is a nice way of thinking about it, but in practice, you might go through your training set and then just take the union of all the words that appear in it.

In some of the tests I've even, by the way, said select these features, but this is one way to think about creating your feature vector, right, as zero and one values, okay? Moving on, yeah. Okay. Ask a question?

Student: I'm getting, kind of, confused on how you compute all those parameters.

Instructor (Andrew Ng): On how I came up with the parameters?

Student: Correct.

Instructor (Andrew Ng): Let's see. So in Naive Bayes, what I need to do – the question was how did I come up with the parameters, right? In Naive Bayes, I need to build a model for PFX given Y and for PFY, right? So this is, I mean, in generous of learning algorithms, I need to come up with models for these. So how'd I model PFY? Well, I just those to model it using a Bernoulli distribution, and so PFY will be parameterized by that, all right?

Student: Okay.

Instructor (Andrew Ng): And then how'd I model PFX given Y? Well, let's keep changing bullets. My model for PFX given Y under the Naive Bayes assumption, I assume that PFX given Y is the product of these probabilities, and so I'm going to need parameters to tell me what's the probability of each word occurring, you know, of each word occurring or not occurring, conditions on the email being spam or not spam email, okay?

Student: How is that Bernoulli?

Instructor (Andrew Ng): Oh, because X is either zero or one, right? By the way I defined the feature vectors, XI is either one or zero, depending on whether words I appear as in the email, right? So by the way I define the feature vectors, XI – the XI is always zero or one. So that by definition, if XI, you know, is either zero or one, then it has to be a Bernoulli distribution, right? If XI would continue as then you might model this as Gaussian and say you end up like we did in Gaussian discriminant analysis. It's just that the way I constructed my features for email, XI is always binary value, and so you end up with a Bernoulli here, okay? All right. I should move on.

So it turns out that this idea almost works. Now, here's the problem. So let's say you complete this class and you start to do, maybe do the class project, and you keep working on your class project for a bit, and it becomes really good, and you want to submit your class project to a conference, right? So, you know, around – I don't know, June every year is the conference deadline for the next conference. It's just the name of the conference; it's an acronym.

And so maybe you send your project partners or senior friends even, and say, "Hey, let's work on a project and submit it to the NIPS conference." And so you're getting these emails with the word "NIPS" in them, which you've probably never seen before, and so a piece of email comes from your project partner, and so you go, "Let's send a paper to the NIPS conference."

And then your stamp classifier will say PFX – let's say NIPS is the 30,000th word in your dictionary, okay? So X30,000 given the 1, given $Y = 1$ will be equal to 0. That's the maximum likelihood of this, right? Because you've never seen the word NIPS before in your training set, so maximum likelihood of the parameter is that probably have seen the word NIPS is zero, and, similarly, you know, in, I guess, non-spam mail, the chance of seeing the word NIPS is also estimated as zero.

So when your spam classifier goes to compute $PFY = 1$ given X, it will compute this right here $\times$ PFY over – well, all right. And so you look at that terms, say, this will be product from $I = 1$ to 50,000, PFXI given Y, and one of those probabilities will be equal to zero because PFX30,000 $= 1$ given $Y = 1$ is equal to zero. So you have a zero in this product, and so the numerator is zero, and in the same way, it turns out the denominator will also be zero, and so you end up with – actually all of these terms end up being zero. So you end up with $PFY = 1$ given X is 0 over $0 + 0$, okay, which is undefined.

And the problem with this is that it's just statistically a bad idea to say that PFX30,000 given Y is 0, right? Just because you haven't seen the word NIPS in your last two months worth of email, it's also statistically not sound to say that, therefore, the chance of ever seeing this word is zero, right?

And so is this idea that just because you haven't seen something before, that may mean that that event is unlikely, but it doesn't mean that it's impossible, and just saying that if you've never seen the word NIPS before, then it is impossible to ever see the word NIPS in future emails; the chance of that is just zero.

So we're gonna fix this, and to motivate the fix I'll talk about – the example we're gonna use is let's say that you've been following the Stanford basketball team for all of their away games, and been, sort of, tracking their wins and losses to gather statistics, and, maybe – I don't know, form a betting pool about whether they're likely to win or lose the next game, okay?

So these are some of the statistics. So on, I guess, the 8th of February last season they played Washington State, and they did not win. On the 11th of February, they play Washington, 22nd they played USC, played UCLA, played USC again, and now you want to estimate what's the chance that they'll win or lose against Louisville, right?

So find the four guys last year or five times and they weren't good in their away games, but it seems awfully harsh to say that – so it seems awfully harsh to say there's zero chance that they'll win in the last – in the 5th game. So here's the idea behind Laplace smoothing which is that we're estimate the probably of Y being equal to one, right? Normally, the maximum likelihood [inaudible] is the number of ones divided by the number of zeros plus the number of ones, okay?

I hope this informal notation makes sense, right? Knowing the maximum likelihood estimate for, sort of, a win or loss for Bernoulli random variable is just the number of ones you saw divided by the total number of examples. So it's the number of zeros you saw plus the number of ones you saw.

So in the Laplace Smoothing we're going to just take each of these terms, the number of ones and, sort of, add one to that, the number of zeros and add one to that, the number of ones and add one to that, and so in our example, instead of estimating the probability of winning the next game to be $0 \div 5 + 0$, we'll add one to all of these counts, and so we say that the chance of their winning the next game is 1/7th, okay? Which is that having seen them lose, you know, five away games in a row, we aren't terribly – we don't think it's terribly likely they'll win the next game, but at least we're not saying it's impossible.

As a historical side note, the Laplace actually came up with the method. It's called the Laplace smoothing after him. When he was trying to estimate the probability that the sun will rise tomorrow, and his rationale was in a lot of days now, we've seen the sun rise, but that doesn't mean we can be absolutely certain the

sun will rise tomorrow. He was using this to estimate the probability that the sun will rise tomorrow. This is, kind of, cool.

So, and more generally, if Y takes on K possible of values, if you're trying to estimate the parameter of the multinomial, then you estimate PFY = 1. Let's see. So the maximum likelihood estimate will be Sum from J = 1 to M, indicator YI = J ÷ M, right? That's the maximum likelihood estimate of a multinomial probability of Y being equal to – oh, excuse me, Y = J. All right. That's the maximum likelihood estimate for the probability of Y = J, and so when you apply Laplace smoothing to that, you add one to the numerator, and add K to the denominator, if Y can take up K possible values, okay?

So for Naive Bayes, what that gives us is – shoot. Right? So that was the maximum likelihood estimate, and what you end up doing is adding one to the numerator and adding two to the denominator, and this solves the problem of the zero probabilities, and when your friend sends you email about the NIPS conference, your spam filter will still be able to make a meaningful prediction, all right? Okay. Shoot. Any questions about this? Yeah?

Student: So that's what doesn't makes sense because, for instance, if you take the games on the right, it's liberal assumptions that the probability of winning is very close to zero, so, I mean, the prediction should be equal to PF, 0.

Instructor (Andrew Ng): Right. I would say that in this case the prediction is 1/7th, right? We don't have a lot of – if you see somebody lose five games in a row, you may not have a lot of faith in them, but as an extreme example, suppose you saw them lose one game, right? It's just not reasonable to say that the chances of winning the next game is zero, but that's what maximum likelihood estimate will say.

Student: Yes.

Instructor (Andrew Ng): And –

Student: In such a case anywhere the learning algorithm [inaudible] or –

Instructor (Andrew Ng): So some questions of, you know, given just five training examples, what's a reasonable estimate for the chance of winning the next game, and 1/7th is, I think, is actually pretty reasonable. It's less than 1/5th for instance. We're saying the chances of winning the next game is less than 1/5th.

It turns out, under a certain set of assumptions I won't go into – under a certain set of Bayesian assumptions about the prior and posterior, this Laplace smoothing actually gives the optimal estimate, in a certain sense I won't go into of what's the chance of winning the next game, and so under a certain assumption about the Bayesian prior on the parameter. So I don't know. It actually seems like a pretty reasonable assumption to me. Although, I should say, it actually turned out –

No, I'm just being mean. We actually are a pretty good basketball team, but I chose a losing streak because it's funnier that way. Let's see. Shoot. Does someone want to – are there other questions about this? No, yeah. Okay. So there's more that I want to say about Naive Bayes, but we'll do that in the next lecture. So let's wrap it for today.

[End of Audio]

Duration: 76 minutes

1.6 Machine Learning Lecture 6[6]

Figure 1.6

MachineLearning-Lecture06

---

[6]This content is available online at <http://cnx.org/content/m45907/1.2/>.

Instructor (Andrew Ng):Okay, good morning. Welcome back. Just one quick announcement for today, which is that this next discussion section as far as for the TA's will mostly be on, sort of, a tutorial on Matlab and Octaves. So I know many of you already have program Matlab or Octave before, but in case not, and you want to, sort of, see along the tutorial on how direct terms and Matlab, please come to this next discussion section.

What I want to do today is continue our discussion of Naïve Bayes, which is the learning algorithm that I started to discuss in the previous lecture and talk about a couple of different event models in Naïve Bayes, and then I'll take a brief digression to talk about neural networks, which is something that I actually won't spend a lot of time on, and then I want to start to talk about support vector machines, and support vector machines is the learning algorithms, the supervised learning algorithm that many people consider the most effective, off-the-shelf supervised learning algorithm. That point of view is debatable, but there are many people that hold that point of view, and we'll start discussing that today, and this will actually take us a few lectures to complete.

So let's talk about Naïve Bayes. To recap from the previous lecture, I started off describing spam classification as the most [inaudible] example for Naïve Bayes in which we would create feature vectors like these, right, that correspond to words in a dictionary. And so, you know, based on what words appear in a piece of email were represented as a feature vector with ones and zeros in the corresponding places, and Naïve Bayes was a generative learning algorithm, and by that I mean it's an algorithm in which we model PFX given Y, and for Naïve Bayes, specifically, we modeled it as product from I equals one to N, PFXI given Y, and also we model PFY, and then we use Bayes Rule, right, to combine these two together, and so our predictions, when you give it a new piece of email you want to tell if it's spam or not spam, you predict RFX over Y, PFY given X, which by Bayes Rule is RFX over Y, PFX given Y, times BY, okay?

So this is Naïve Bayes, and just to draw attention to two things, one is that in this model, each of our features were zero, one, so indicating whether different words appear, and the length or the feature vector was, sort of, the length N of the feature vector was the number of words in the dictionary. So it might be on this version on the order of 50,000 words, say.

What I want to do now is describe two variations on this algorithm. The first one is the simpler one, which it's just a generalization to if XI takes on more values. So, you know, one thing that's commonly done is to apply Naïve Bayes to problems where some of these features, XI, takes on K values rather than just two values, and in that case, you actually build, sort of, a very similar model where PFX given Y is really the same thing, right, where now these are going to be multinomial probabilities rather than Bernoulli's because the XI's can, maybe, take on up to K values.

It turns out, the situation where − one situation where this arises very commonly is if you have a feature that's actually continuous valued, and you choose to dispertise it, and you choose to take a continuous value feature and dispertise it into a finite set of K values, and so it's a perfect example if you remember our very first supervised learning problem of predicting the price of houses. If you have the classification problem on these houses, so based on features of a house, and you want to predict whether or not the house will be sold in the next six months, say.

That's a classification problem, and once you use Naïve Bayes, then given a continuous value feature like the living area, you know, one pretty common thing to do would be take the continuous value living area and just dispertise it into a few − discreet buckets, and so depending on whether the living area of the house is less than 500 square feet or between 1,000 and 1500 square feet, and so on, or whether it's greater than 2,000 square feet, you choose the value of the corresponding feature, XI, to be one, two, three, or four, okay? So that was the first variation or generalization of Naïve Bayes I wanted to talk about. I should just check; are there questions about this? Okay. Cool. And so it turns out that in practice, it's fairly common to use about ten buckets to dispertise a continuous value feature. I drew four here only to save on writing.

The second and, sort of, final variation that I want to talk about for Naïve Bayes is a variation that's specific to classifying text documents, or, more generally, for classifying sequences. So the text document, like a piece of email, you can think of as a sequence of words and you can apply this, sort of, model I'm about to describe to classifying other sequences as well, but let me just focus on text, and here's the idea.

So the Naïve Bayes algorithm as I've described it so far, right, given a piece of email, we were representing

it using this binary vector value representation, and one of the things that this loses, for instance, is the number of times that different words appear, all right? So, for example, if some word appears a lot of times, and you see the word, you know, "buy" a lot of times. You see the word "Viagra"; it seems to be a common email example. You see the word Viagra a ton of times in the email, it is more likely to be spam than it appears, I guess, only once because even once, I guess, is enough.

So let me just try a different, what's called an event model for Naïve Bayes that will take into account the number of times a word appears in the email, and to give this previous model a name as well this particular model for text classification is called the Multivariate Bernoulli Event Model. It's not a great name. Don't worry about what the name means. It refers to the fact that there are multiple Bernoulli random variables, but it's really – don't worry about what the name means.

In contrast, what I want to do now is describe a different representation for email in terms of the feature vector, and this is called the Multinomial Event Model, and, again, there is a rationale behind the name, but it's slightly cryptic, so don't worry about why it's called the Multinomial Event Model; it's just called that. And here's what we're gonna do, given a piece of email, I'm going to represent my email as a feature vector, and so my IF training example, XI will be a feature vector, XI sub group one, XI sub group two, XI subscript NI where NI is equal to the number of words in this email, right?

So if one of my training examples is an email with 300 words in it, then I represent this email via a feature vector with 300 elements, and each of these elements of the feature vector – lets see. Let me just write this as X subscript J. These will be an index into my dictionary, okay? And so if my dictionary has 50,000 words, then each position in my feature vector will be a variable that takes on one of 50,000 possible values corresponding to what word appeared in the J position of my email, okay?

So, in other words, I'm gonna take all the words in my email and you have a feature vector that just says which word in my dictionary was each word in the email, okay? So a different definition for NI now, NI now varies and is different for every training example, and this XJ is now indexed into the dictionary. You know, the components of the feature vector are no longer binary random variables; they're these indices in the dictionary that take on a much larger set of values.

And so our generative model for this will be that the joint distribution over X and Y will be that, where again N is now the length of the email, all right? So the way to think about this formula is you imagine that there was some probably distribution over emails. There's some random distribution that generates the emails, and that process proceeds as follows: First, Y is chosen, first the class label. Is someone gonna send you spam email or not spam emails is chosen for us.

So first Y, the random variable Y, the class label of spam or not spam is generated, and then having decided whether they sent you spam or not spam, someone iterates over all 300 positions of the email, or 300 words that are going to compose them as email, and would generate words from some distribution that depends on whether they chose to send you spam or not spam. So if they sent you spam, they'll send you words – they'll tend to generate words like, you know, buy, and Viagra, and whatever at discounts, sale, whatever. And if somebody chose to send you not spam, then they'll send you, sort of, the more normal words you get in an email, okay?

So, sort of, just careful, right? XI here has a very different definition from the previous event model, and N has a very different definition from the previous event model. And so the parameters of this model are – let's see. Phi subscript K given Y equals one, which is the probability that, you know, conditioned on someone deciding to spend you spam, what's the probability that the next word they choose to email you in the spam email is going to be word K, and similarly, you know, sort of, same thing – well, I'll just write it out, I guess – and Phi Y and just same as before, okay?

So these are the parameters of the model, and given a training set, you can work out the maximum likelihood estimates of the parameters. So the maximum likelihood estimate of the parameters will be equal to – and now I'm gonna write one of these, you know, big indicator function things again. It'll be a sum over your training sets indicator whether that was spam times the sum over all the words in that email where N subscript I is the number of words in email I in your training set, times indicator XIJ, SK times that I, okay?

So the numerator says sum over all your emails and take into account all the emails that had class label

one, take into account only of the emails that were spam because if Y equals zero, then this is zero, and this would go away, and then times sum over all the words in your spam email, and it counts up the number of times you observed the word K in your spam emails. So, in other words, the numerator is look at all the spam emails in your training set and count up the total number of times the word K appeared in this email. The denominator then is sum over I into our training set of whenever one of your examples is spam, you know, sum up the length of that spam email, and so the denominator is the total length of all of the spam emails in your training set.

And so the ratio is just out of all your spam emails, what is the fraction of words in all your spam emails that were word K, and that's your estimate for the probability of the next piece of spam mail generating the word K in any given position, okay? At the end of the previous lecture, I talked about LaPlace smoothing, and so when you do that as well, you add one to the numerator and K to the denominator, and this is the LaPlace smoothed estimate of this parameter, okay? And similarly, you do the same thing for – and you can work out the estimates for the other parameters yourself, okay? So it's very similar. Yeah?

Student:I'm sorry. On the right on the top, I was just wondering what the X of I is, and what the N of –

Instructor (Andrew Ng):Right. So in this second event model, the definition for XI and the definition for N are different, right? So here – well, this is for one example XY. So here, N is the number of words in a given email, right? And if it's the I email subscripting then this N subscript I, and so N will be different for different training examples, and here XI will be, you know, these values from 1 to 50,000, and XI is essentially the identity of the Ith word in a given piece of email, okay? So that's why this is grouping, or this is a product over all the different words of your email of their probability the Ith word in your email, conditioned on Y. Yeah?

Student:[Off mic].

Instructor (Andrew Ng):Oh, no, actually, you know what, I apologize. I just realized that overload the notation, right, and I shouldn't have used K here. Let me use a different alphabet and see if that makes sense; does that make sense? Oh, you know what, I'm sorry. You're absolutely right. Thank you. All right. So in LaPlace smoothing, that shouldn't be K. This should be, you know, 50,000, if you have 50,000 words in your dictionary. Yeah, thanks. Great. I stole notation from the previous lecture and didn't translate it properly. So LaPlace smoothing, right, this is the number of possible values that the random variable XI can take on. Cool. Raise your hand if this makes sense? Okay. Some of you, are there more questions about this? Yeah.

Student: On LaPlace smoothing, the denominator and the plus A is the number of values that Y could take?

Instructor (Andrew Ng):Yeah, let's see. So LaPlace smoothing is a method to give you, sort of, hopefully, better estimates of their probability distribution over a multinomial, and so was I using X to Y in the previous lecture? So in trying to estimate the probability over a multinomial – I think X and Y are different. I think – was it X or Y? I think it was X, actually. Well – oh, I see, right, right. I think I was using a different definition for the random variable Y because suppose you have a multinomial random variable, X which takes on – let's use a different alphabet. Suppose you have a multinomial random variable X which takes on L different values, then the maximum likelihood estimate for the probability of X, PFX equals K, will be equal to, right, the number of observations. The maximum likelihood estimate for the probability of X being equal to K will be the number of observations of X equals K divided by the total number of observations of X, okay? So that's the maximum likelihood estimate. And to add LaPlace smoothing to this, you, sort of, add one to the numerator, and you add L to the denominator where L was the number of possible values that X can take on. So, in this case, this is a probability that X equals K, and X can take on 50,000 values if 50,000 is the length of your dictionary; it may be something else, but that's why I add 50,000 to the denominator. Are there other questions? Yeah.

Student: Is there a specific definition for a maximum likelihood estimation of a parameter? We've talked about it a couple times, and all the examples make sense, but I don't know what the, like, general formula for it is.

Instructor (Andrew Ng):I see. Yeah, right. So the definition of maximum likelihood – so the question

is what's the definition for maximum likelihood estimate? So actually in today's lecture and the previous lecture when I talk about Gaussian Discriminant Analysis I was, sort of, throwing out the maximum likelihood estimates on the board without proving them. The way to actually work this out is to actually write down the likelihood.

So the way to figure out all of these maximum likelihood estimates is to write down the likelihood of the parameters, phi K given Y being zero, phi Y, right? And so given a training set, the likelihood, I guess, I should be writing log likelihood will be the log of the product of I equals one to N, PFXI, YI, you know, parameterized by these things, okay? Where PFXI, YI, right, is given by NI, PFX, YJ given YI. They are parameterized by − well, I'll just drop the parameters to write this more simply − oh, I just put it in − times PFYI, okay?

So this is my log likelihood, and so the way you get the maximum likelihood estimate of the parameters is you − so if given a fixed training set, given a set of fixed IYI's, you maximize this in terms of these parameters, and then you get the maximum likelihood estimates that I've been writing out. So in a previous section of today's lecture I wrote out some maximum likelihood estimates for the Gaussian Discriminant Analysis model, and for Naïve Bayes, and then this − I didn't prove them, but you get to, sort of, play with that yourself in the homework problem as well and for one of these models, and you'll be able to verify that when you maximize the likelihood and maximize the log likelihood that hopefully you do get the same formulas as what I was drawing up on the board, but a way is to find the way these are derived is by maximizing this, okay? Cool.

All right. So that wraps up what I wanted to say about − oh, so that, more or less, wraps up what I wanted to say about Naïve Bayes, and it turns out that for text classification, the Naïve Bayes algorithm with this second event model, the last Naïve Bayes model I presented with the multinomial event model, it turns out that almost always does better than the first Naïve Bayes model I talked about when you're applying it to the specific case − to the specific of text classification, and one of the reasons is hypothesized for this is that this second model, the multinomial event model, takes into account the number of times a word appears in a document, whereas the former model doesn't.

I should say that in truth that actually turns out not to be completely understood why the latter model does better than the former one for text classification, and, sort of, researchers are still debating about it a little bit, but if you ever have a text classification problem, you know, Naïve Bayes Classify is probably not, by far, the best learning algorithm out there, but it is relatively straightforward to implement, and it's a very good algorithm to try if you have a text classification problem, okay? Still a question? Yeah.

Student: So the second model is still positioning a variant, right? It doesn't actually care where the words are.

Instructor (Andrew Ng): Yes, all right.

Student: And, I mean, X variable, if my model like you had exclamation in, does that usually do better if you have enough data?

Instructor (Andrew Ng): Yeah, so the question is, sort of, the second model, right? The second model, the multinomial event model actually doesn't care about the ordering of the words. You can shuffle all the words in the email, and it does exactly the same thing. So in natural language processing, there's actually another name; it's called a Unique Grand Model in natural language processing, and there's some other models like, sort of, say, higher order markup models that take into account some of the ordering of the words. It turns out that for text classification, the models like the bigram models or trigram models, I believe they do only very slightly better, if at all, but that's when you're applying them to text classification, okay?

All right. So the next thing I want to talk about is to start again to discussion of non-linear classifiers. So it turns out − well, and so the very first classification algorithm we talked about was logistic regression, which had the forming form for hypothesis, and you can think of this as predicting one when this estimator probability is greater or equal to 0.5 and predicting zero, right, when this is less than 0.5, and given a training set, right? Logistic regression will maybe do grade and descends or something or use Newton's method to find a straight line that reasonably separates the positive and negative classes.

But sometimes a data set just can't be separated by a straight line, so is there an algorithm that will let you start to learn these sorts of non-linear division boundaries? And so how do you go about getting

a non-linear classifier? And, by the way, one cool result is that remember when I said – when we talked about generative learning algorithms, I said that if you assume Y given X is exponential family, right, with parameter A, and if you build a generative learning algorithm using this, right, plus one, if this is A to one. This is exponential family with natural parameter A to zero, right.

I think when we talked about Gaussian Discriminant Analysis, I said that if this holds true, then you end up with a logistic posterior. It actually turns out that a Naïve Bayes model actually falls into this as well. So the Naïve Bayes model actually falls into this exponential family as well, and, therefore, under the Naïve Bayes model, you're actually using this other linear classifier as well, okay?

So the question is how can you start to get non-linear classifiers? And I'm going to talk about one method today which is – and we started to talk about it very briefly which is taking a simpler algorithm like logistic regression and using it to build up to more complex non-linear classifiers, okay? So to motivate this discussion, I'm going to use the little picture – let's see. So suppose you have features X1, X2, and X3, and so by convention, I'm gonna follow our earlier convention that X0 is set to one, and so I'm gonna use a little diagram like this to denote our logistic regression unit, okay?

So think of a little picture like that, you know, this little circle as denoting a computation note that takes this input, you know, several features and then it outputs another number that's X subscript theta of X, given by a sigmoid function, and so this little computational unit – well, will have parameters theta.

Now, in order to get non-linear division boundaries, all we need to do – well, at least one thing to do is just come up with a way to represent hypotheses that can output non-linear division boundaries, right, and so this is – when you put a bunch of those little pictures that I drew on the previous board, you can then get what's called a Neural Network in which you think of having my features here and then I would feed them to say a few of these little sigmoidal units, and these together will feed into yet another sigmoidal unit, say, which will output my final output H subscript theta of X, okay? And just to give these things names, let me call the values output by these three intermediate sigmoidal units; let me call them A1, A2, A3.

And let me just be completely concrete about what this formula represents, right? So each of these units in the middle will have their own associated set of parameters, and so the value A1 will be computed as G of X transpose, and then some set of parameters, which I'll write as theta one, and similarly, A2 will be computed as G of X transpose theta two, and A3 will be G of X transpose, theta three, where G is the sigmoid function, all right? So G of Z, and then, finally, our hypothesis will output G of A transpose theta four, right? Where, you know, this A vector is a vector of A1, A2, A3. We can append another one to it at first if you want, okay?

Let me just draw up here this – I'm sorry about the cluttered board. And so H subscript theta of X, this is a function of all the parameters theta one through theta four, and so one way to learn parameters for this model is to write down the cost function, say, J of theta equals one-half sum from Y equals one to M, YI minus H subscript theta of XI squared, say. Okay, so that's our familiar quadratic cost function, and so one way to learn the parameters of an algorithm like this is to just use gradient interscent to minimize J of theta as a function of theta, okay? See, in the phi gradient descent to minimize this square area, which stated differently means you use gradient descent to make the predictions of your neural network as close as possible to what you observed as the labels in your training set, okay?

So it turns out green descent on this neural network is a specific name, the algorithm that implements grand descent is called back propagation, and so if you ever hear that all that means is – it just means gradient interscent on a cost function like this or a variation of this on the neural network that looks like that, and – well, this algorithm actually has some advantages and disadvantages, but let me actually show you. So, let's see.

One of the interesting things about the neural network is that you can look at what these intermediate notes are computing, right? So this neural network has what's called a hidden layer before you then have the output layer, and, more generally, you can actually have inputs feed into these computation units, feed into more layers of computation units, to even more layers, to more layers, and then finally you have an output layer at the end

And one cool thing you can do is look at all of these intermediate units, look at these units and what's called a hidden layer of the neural network. Don't worry about why it's called that. Look at computations

of the hidden unit and ask what is the hidden unit computing the neural network? So to, maybe, get a better sense of neural networks might be doing, let me show you a video – I'm gonna switch to the laptop – this is made by a friend, Yann LeCun who's currently a professor at New York University. Can I show a video on the laptop?

So let me show you a video from Yann LeCun on a neural network that he developed for Hammerton Digit Recognition. There was one other thing he did in this neural network that I'm not gonna talk about called a Convolutional Neural Network that – well, his system is called LeNet, and let's see. Would you put on the laptop display? Hum, actually maybe if – or you can just put on the screen on the side; that would work too if the big screen isn't working. Let's see. I'm just trying to think, okay, how do I keep you guys entertained while we're waiting for the video to come on?

Well, let me say a few more things about neural network. So it turns out that when you write a quadratic cost function like I wrote down on the chalkboard just now, it turns out that unlike logistic regression, that will almost always respond to non-convex optimization problem, and so whereas for logistic regression if you run gradient descent or Newton's method or whatever, you converse the global optimer. This is not true for neural networks. In general, there are lots of local optimer and, sort of, much harder optimization problem.

So neural networks, if you're, sort of, familiar with them, and you're good at making design choices like what learning rate to use, and how many hidden units to use, and so on, you can, sort of, get them to be fairly effective, and there's, sort of, often ongoing debates about, you know, is this learning algorithm better, or is that learning algorithm better? The vast majority of machine learning researchers today seem to perceive support vector machines, which is what I'll talk about later, to be a much more effective off-the-shelf learning algorithm than neural networks. This point of view is contested a bit, but so neural networks are not something that I personally use a lot right now because there's a hard optimization problem and you should do so often verge, and it actually, sort of works. It, sort of, works reasonably well. It's just because this is fairly complicated, there's not an algorithm that I use commonly or that my friends use all time. Oh, cool.

So but let me just go and show you an example of neural network, which was for many years, you know, the most effective learning algorithm before support vector machines were invented. So here's Yann LeCun's video, and – well, there's actually audio on this too, the soundboard. So I'll just tell you what's happening. What you're seeing is a trained neural network, and this display where my mouse pointer is pointing, right, this big three there is the input to the neural network.

So you're showing the neural network this image, and it's trying to recognize what is this. The final answer output by the neural network is this number up here, right below where it says LeNet-5, and the neural network correctly recognizes this image as a three, and if you look to the left of this image, what's interesting about this is the display on the left portion of this is actually showing the intermediate computations of the neural network. In other words, it's showing you what are the hidden layers of the neural network computing.

And so, for example, if you look at this one, the third image down from the top, this seems to be computing, you know, certain edges into digits, right? We're just computing digits on the right-hand side of the bottom or something of the input display of the input image, okay? So let me just play this video, and you can see some of the inputs and outputs of the neural network, and those are very different fonts. There's this robustness to noise. All right. Multiple digits, that's, kind of, cool. All right.

So, just for fun, let me show you one more video, which was – let's see. This is another video from the various CV's, the machine that changed the world, which was produced by WGBH Television in corporation with British Foreclass Incorporation, and it was aired on PBS a few years ago, I think. I want to show you a video describing the NETtalk Neural Network, which was developed by Terry Sejnowski; he's a researcher. And so NETtalk was actually one of the major milestones in the history of neural network, and this specific application is getting the neural network to read text.

So, in other words, can you show a piece of English to a computer and have the computer read, sort of, verbally produce sounds that could respond to the reading of the text. And it turns out that in the history of AI and the history of machine learning, this video created a lot of excitement about neural networks and about machine learning. Part of the reason was that Terry Sejnowski had the foresight to choose to use, in his video, a child-like voice talking about visiting your grandmother's house and so on.

You'll see it in a second, and so this really created the perception of – created the impression of the neural network being like a young child learning how to speak, and talking about going to your grandmothers, and so on. So this actually helped generate a lot of excitement within academia and outside academia on neural networks, sort of, early in the history of neural networks. I'm just gonna show you the video.

[Begin Video] You're going to hear first what the network sounds like at the very beginning of the training, and it won't sound like words, but it'll sound like attempts that will get better and better with time. [Computer's voice] The network takes the letters, say the phrase, "grandmother's house," and makes a random attempt at pronouncing it. [Computer's voice] Grandmother's house. The phonetic difference between the guess and the right pronunciation is sent back through the network. [Computer's voice] Grandmother's house. By adjusting the connection strengths after each attempt, the net slowly improves.

And, finally, after letting it train overnight, the next morning it sounds like this: Grandmother's house, I'd like to go to my grandmother's house. Well, because she gives us candy. Well, and we – NETtalk understands nothing about the language. It is simply associating letters with sounds. [End Video]

All right. So at the time this was done, I mean, this is an amazing piece of work. I should say today there are other text to speech systems that work better than what you just saw, and you'll also appreciate getting candy from your grandmother's house is a little bit less impressive than talking about the Dow Jones falling 15 points, and profit taking, whatever. So but I wanted to show that just because that was another cool, major landmark in the history of neural networks. Okay. So let's switch back to the chalkboard, and what I want to do next is tell you about Support Vector Machines, okay?

That, sort of, wraps up our discussion on neural networks. So I started off talking about neural networks by motivating it as a way to get us to output non-linear classifiers, right? I don't really approve of it. It turns out that you'd be able to come up with non-linear division boundaries using a neural network like what I drew on the chalkboard earlier.

Support Vector Machines will be another learning algorithm that will give us a way to come up with non-linear classifiers. There's a very effective, off-the-shelf learning algorithm, but it turns out that in the discussion I'm gonna – in the progression and development I'm gonna pursue, I'm actually going to start off by describing yet another class of linear classifiers with linear division boundaries, and only be later, sort of, in probably the next lecture or the one after that, that we'll then take the support vector machine idea and, sort of, do some clever things to it to make it work very well to generate non-linear division boundaries as well, okay? But we'll actually start by talking about linear classifiers a little bit more.

And to do that, I want to convey two intuitions about classification. One is you think about logistic regression; we have this logistic function that was outputting the probability that Y equals one, and it crosses this line at zero. So when you run logistic regression, I want you to think of it as an algorithm that computes theta transpose X, and then it predicts one, right, if and only if, theta transpose X is greater than zero, right? IFF stands for if and only if. It means the same thing as a double implication, and it predicts zero, if and only if, theta transpose X is less than zero, okay?

So if it's the case that theta transpose X is much greater than zero, the double greater than sign means these are much greater than, all right. So if theta transpose X is much greater than zero, then, you know, think of that as a very confident prediction that Y is equal to one, right? If theta transpose X is much greater than zero, then we're gonna predict one then moreover we're very confident it's one, and the picture for that is if theta transpose X is way out here, then we're estimating that the probability of Y being equal to one on the sigmoid function, it will be very close to one. And, in the same way, if theta transpose X is much less than zero, then we're very confident that Y is equal to zero.

So wouldn't it be nice – so when we fit logistic regression of some of the classifiers is your training set, then so wouldn't it be nice if, right, for all I such that Y is equal to one. We have theta transpose XI is much greater than zero, and for all I such that Y is equal to zero, we have theta transpose XI is much less than zero, okay? So wouldn't it be nice if this is true? That, essentially, if our training set, we can find parameters theta so that our learning algorithm not only makes correct classifications on all the examples in a training set, but further it's, sort of, is very confident about all of those correct classifications. This is the first intuition that I want you to have, and we'll come back to this first intuition in a second when we talk about functional margins, okay? We'll define this later.

The second intuition that I want to convey, and it turns out for the rest of today's lecture I'm going to assume that a training set is linearly separable, okay? So by that I mean for the rest of today's lecture, I'm going to assume that there is indeed a straight line that can separate your training set, and we'll remove this assumption later, but just to develop the algorithm, let's take away the linearly separable training set. And so there's a sense that out of all the straight lines that separate the training set, you know, maybe that straight line isn't such a good one, and that one actually isn't such a great one either, but maybe that line in the middle is a much better linear separator than the others, right?

And one reason that when you and I look at it this one seems best is because this line is just further from the data, all right? That is separates the data with a greater distance between your positive and your negative examples and division boundary, okay? And this second intuition, we'll come back to this shortly, about this final line that I drew being, maybe, the best line this notion of distance from the training examples. This is the second intuition I want to convey, and we'll formalize it later when we talk about geometric margins of our classifiers, okay?

So in order to describe support vector machine, unfortunately, I'm gonna have to pull a notation change, and, sort of, unfortunately, it, sort of, was impossible to do logistic regression, and support vector machines, and all the other algorithms using one completely consistent notation, and so I'm actually gonna change notations slightly for linear classifiers, and that will actually make it much easier for us – that'll make it much easier later today and in next week's lectures to actually talk about support vector machine.

But the notation that I'm gonna use for the rest of today and for most of next week will be that my B equals Y, and instead of be zero, one, they'll be minus one and plus one, and a development of a support vector machine we will have H, have a hypothesis output values to the either plus one or minus one, and so we'll let G of Z be equal to one if Z is greater or equal to zero, and minus one otherwise, right? So just rather than zero and one, we change everything to plus one and minus one.

And, finally, whereas previously I wrote G subscript theta of X equals G of theta transpose X and we had the convention that X zero is equal to one, right? And so X is an RN plus one. I'm gonna drop this convention of letting X zero equals a one, and letting X be an RN plus one, and instead I'm going to parameterize my linear classifier as H subscript W, B of X equals G of W transpose X plus B, okay? And so B just now plays the role of theta zero, and W now plays the role of the rest of the parameters, theta one through theta N, okay? So just by separating out the interceptor B rather than lumping it together, it'll make it easier for us to develop support vector machines. So – yes.

Student:[Off mic].

Instructor (Andrew Ng): Oh, yes. Right, yes. So W is – right. So W is a vector in RN, and X is now a vector in RN rather than N plus one, and a lowercase b is a real number. Okay.

Now, let's formalize the notion of functional margin and germesh margin. Let me make a definition. I'm going to say that the functional margin of the hyper plane WB with respect to a specific training example, XIYI is – WRT stands for with respect to – the function margin of a hyper plane WB with respect to a certain training example, XIYI has been defined as Gamma Hat I equals YI times W transpose XI plus B, okay?

And so a set of parameters, W, B defines a classifier – it, sort of, defines a linear separating boundary, and so when I say hyper plane, I just mean the decision boundary that's defined by the parameters W, B. You know what, if you're confused by the hyper plane term, just ignore it. The hyper plane of a classifier with parameters W, B with respect to a training example is given by this formula, okay? And interpretation of this is that if YI is equal to one, then for each to have a large functional margin, you want W transpose XI plus B to be large, right? And if YI is equal minus one, then in order for the functional margin to be large – we, sort of, want the functional margins to large, but in order for the function margins to be large, if YI is equal to minus one, then the only way for this to be big is if W transpose XI plus B is much less than zero, okay?

So this captures the intuition that we had earlier about functional margins – the intuition we had earlier that if YI is equal to one, we want this to be big, and if YI is equal to minus one, we want this to be small, and this, sort of, practice of two cases into one statement that we'd like the functional margin to be large. And notice this is also that so long as YI times W transpose XY plus B, so long as this is greater than zero,

that means we classified it correctly, okay?

And one more definition, I'm going to say that the functional margin of a hyper plane with respect to an entire training set is going to define gamma hat to be equal to min over all your training examples of gamma hat, I, right? So if you have a training set, if you have just more than one training example, I'm going to define the functional margin with respect to the entire training set as the worst case of all of your functional margins of the entire training set. And so for now we should think of the first function like an intuition of saying that we would like the function margin to be large, and for our purposes, for now, let's just say we would like the worst-case functional margin to be large, okay? And we'll change this a little bit later as well.

Now, it turns out that there's one little problem with this intuition that will, sort of, edge us later, which it actually turns out to be very easy to make the functional margin large, all right? So, for example, so as I have a classifiable parameters W and B. If I take W and multiply it by two and take B and multiply it by two, then if you refer to the definition of the functional margin, I guess that was what? Gamma I, gamma hat I equals YI times W times transpose B. If I double W and B, then I can easily double my functional margin.

So this goal of making the functional margin large, in and of itself, isn't so useful because it's easy to make the functional margin arbitrarily large just by scaling other parameters. And so maybe one thing we need to do later is add a normalization condition. For example, maybe we want to add a normalization condition that de-norm, the alter-norm of the parameter W is equal to one, and we'll come back to this in a second. All right. And then so –

Okay. Now, let's talk about – see how much time we have, 15 minutes. Well, see, I'm trying to decide how much to try to do in the last 15 minutes. Okay. So let's talk about the geometric margin, and so the geometric margin of a training example – [inaudible], right? So the division boundary of my classifier is going to be given by the plane W transpose X plus B is equal to zero, okay? Right, and these are the X1, X2 axis, say, and we're going to draw relatively few training examples here. Let's say I'm drawing deliberately few training examples so that I can add things to this, okay?

And so assuming we classified an example correctly, I'm going to define the geometric margin as just a geometric distance between a point between the training example – yeah, between the training example XI, YI and the distance given by this separating line, given by this separating hyper plane, okay? That's what I'm going to define the geometric margin to be.

And so I'm gonna do some algebra fairly quickly. In case it doesn't make sense, and read through the lecture notes more carefully for details. Sort of, by standard geometry, the normal, or in other words, the vector that's 90 degrees to the separating hyper plane is going to be given by W divided by the norm of W; that's just how planes and high dimensions work. If this stuff – some of this you have to use, take a look t the lecture notes on the website.

And so let's say this distance is gamma I, okay? And so I'm going to use the convention that I'll put a hat on top where I'm referring to functional margins, and no hat on top for geometric margins. So let's say geometric margin, as this example, is gamma I. That means that this point here, right, is going to be XI minus gamma I times W over normal W, okay? Because W over normal W is the unit vector, is the length one vector that is normal to the separating hyper plane, and so when we subtract gamma I times the unit vector from this point, XI, or at this point here is XI. So XI minus, you know, this little vector here is going to be this point that I've drawn as a heavy circle, okay? So this heavy point here is XI minus this vector, and this vector is gamma I time W over norm of W, okay?

And so because this heavy point is on the separating hyper plane, right, this point must satisfy W transpose times that point equals zero, right? Because all points X on the separating hyper plane satisfy the equation W transpose X plus B equals zero, and so this point is on the separating hyper plane, therefore, it must satisfy W transpose this point – oh, excuse me. Plus B is equal to zero, okay? Raise your hand if this makes sense so far? Oh, okay. Cool, most of you, but, again, I'm, sort of, being slightly fast in this geometry. So if you're not quite sure why this is a normal vector, or how I subtracted this, or whatever, take a look at the details in the lecture notes.

And so what I'm going to do is I'll just take this equation, and I'll solve for gamma, right? So this equation I just wrote down, solve this equation for gamma or gamma I, and you find that – you saw that

previous equation from gamma I – well, why don't I just do it? You have W transpose XI plus B equals gamma I times W transpose W over norm of W; that's just equal to gamma times the norm of W because W transpose W is the norm of W squared, and, therefore, gamma is just – well, transpose X equals, okay? And, in other words, this little calculation just showed us that if you have a training example XI, then the distance between XI and the separating hyper plane defined by the parameters W and B can be computed by this formula, okay?

So the last thing I want to do is actually take into account the sign of the – the correct classification of the training example. So I've been assuming that we've been classifying an example correctly. So, more generally, to find the geometric margin of a training example to be gamma I equals YI times that thing on top, okay? And so this is very similar to the functional margin, except for the normalization by the norm of W, and so as before, you know, this says that so long as – we would like the geometric margin to be large, and all that means is that so long as we're classifying the example correctly, we would ideally hope of the example to be as far as possible from the separating hyper plane, so long as it's on the right side of the separating hyper plane, and that's what YI multiplied into this does.

And so a couple of easy facts, one is if the norm of W is equal to one, then the functional margin is equal to the geometric margin, and you see that quite easily, and, more generally, the geometric margin is just equal to the functional margin divided by the norm of W, okay? Let's see, okay. And so one final definition is so far I've defined the geometric margin with respect to a single training example, and so as before, I'll define the geometric margin with respect to an entire training set as gamma equals min over I of gamma I, all right?

And so the maximum margin classifier, which is a precursor to the support vector machine, is the learning algorithm that chooses the parameters W and B so as to maximize the geometric margin, and so I just write that down. The maximum margin classified poses the following optimization problem. It says choose gamma, W, and B so as to maximize the geometric margin, subject to that YI times – well, this is just one way to write it, subject to – actually, do I write it like that? Yeah, fine. There are several ways to write this, and one of the things we'll do next time is actually – I'm trying to figure out if I can do this in five minutes. I'm guessing this could be difficult.

Well, so this maximizing your classifier is the maximization problem over parameter gamma W and B, and for now, it turns out that the geometric margin doesn't change depending on the norm of W, right? Because in the definition of the geometric margin, notice that we're dividing by the norm of W anyway. So you can actually set the norm of W to be anything you want, and you can multiply W and B by any constant; it doesn't change the geometric margin. This will actually be important, and we'll come back to this later. Notice that you can take the parameters WB, and you can impose any normalization constant to it, or you can change W and B by any scaling factor and replace them by ten W and ten B whatever, and it does not change the geometric margin, okay?

And so in this first formulation, I'm just gonna impose a constraint and say that the norm of W was one, and so the function of the geometric margins will be the same, and then we'll say maximize the geometric margins subject to – you maximize gamma subject to that every training example must have geometric margin at least gamma, and this is a geometric margin because when the norm of W is equal to one, then the functional of the geometric margin are identical, okay?

So this is the maximum margin classifier, and it turns out that if you do this, it'll run, you know, maybe about as well as a – maybe slight – maybe comparable to logistic regression, but it turns out that as we develop this algorithm further, there will be a clever way to allow us to change this algorithm to let it work in infinite dimensional feature spaces and come up with very efficient non-linear classifiers. So there's a ways to go before we turn this into a support vector machine, but this is the first step. So are there questions about this? Yeah.

Student:[Off mic].

Instructor (Andrew Ng): For now, let's just say you're given a fixed training set, and you can't – yeah, for now, let's just say you're given a fixed training set, and the scaling of the training set is not something you get to play with, right? So everything I've said is for a fixed training set, so that you can't change the X's, and you can't change the Y's. Are there other questions?

Okay. So all right. Next week we will take this, and we'll talk about authorization algorithms, and work our way towards turning this into one of the most effective off-the-shelf learning algorithms, and just a final reminder again, this next discussion session will be on Matlab and Octaves. So show up for that if you want to see a tutorial. Okay. See you guys in the next class.

[End of Audio]

Duration: 74 minutes

1.7 Machine Learning Lecture 7[7]

Figure 1.7

MachineLearning-Lecture07

Instructor (Andrew Ng): Good morning. So just a few quick announcements. One is for the SCPD students − that would be for the students taking the classes remotely − to turn in the process solutions due this Wednesday, please fax the solutions to us at − so the fax number written at the top of the Problem Set 1. And in particular, please do not use the SCPD [inaudible], since that usually takes me longer for your solutions to get to us.

And everyone else, if you're not an SCPD student, then please only turn in hard copies − the physical paper copies of your solutions from Set 1. And please don't email them to us or send them by fax unless you're an SCPD student.

Also, as you know, project proposals are due this Friday. And so in my last few office hours, you should have lively discussions with people about [inaudible] ideas.

This Wednesday, immediately after class, so this Wednesday, starting about, I guess 10:45, right after class, I'll be holding extra office hours in case any of you want to discuss project ideas some more before the proposals are due on Friday, okay?

But is this loud enough? Can people in the back hear me? Is this okay?

Student:[Inaudible] louder, maybe?

Instructor (Andrew Ng):[Inaudible] turn up the volume, [Inaudible]? Is this okay? Testing, testing. It's better? Okay, great.

So that was it for the administrative announcements.

So welcome back. And what I wanna do today is continue our discussion on support vector machines. And in particular, I wanna talk about the optimal margin classifier. Then I wanna take a brief digression and talk about primal and duo optimization problems, and in particular, what's called the KKT conditions. And then we'll derive the duo to the optimization problem that I had posed earlier.

And that will lead us into a discussion of kernels, which I won't really − which we just get to say couple words about, but which I'll do probably only in the next lecture.

And as part of today's lecture, I'll spend some time talking about optimization problems. And in the little time I have today, I won't really be able to do this topic justice. I wanna talk about convex optimization and do that topic justice. And so at this week's discussion session, the TAs will have more time − will teach a discussion session − focus on convex optimization − sort of very beautiful and useful theory. So you want to learn more about that, listen to this Friday's discussion session.

Just to recap what we did in the previous lecture, as we were beginning on developing on support vector machines, I said that a hypothesis represented as H sub [inaudible] wb as g of w transpose [inaudible] x +

---

[7]This content is available online at <http://cnx.org/content/m45896/1.2/>.

b, where g will be +1 or -1, depending on whether z is greater than 0. And I said that in our development of support vector machines, we'll use – we'll change the convention of letting y be +1, -1 to note the class labels.

So last time, we also talked about the functional margin, which was this thing, gamma hat i. And so we had the intuition that the if functional margin is a large positive number, then that means that we are classifying a training example correctly and very confidently.

So yi is +1. We would like w transpose xi + b to be very large. And it makes i – if, excuse me, if yi is -1, then we'd w transpose xi + b to be a large negative number. So we'd sort of like functional margins to be large.

We also said – functional margin is a strange property – that you can increase functional margin just by, say, taking your parameters, w and b, and multiplying them by 2.

And then we also defined the geometric margin, which was that we just – essentially, the functional margin divided by the normal w.

And so the geometric margin had the interpretation as being – I'll give you a few examples. The geometric margin, for example, is – has the interpretation as a distance between a training example and a hyperplane.

And it'll actually be a sin distance, so that this distance will be positive if you're classifying the example correctly. And if you misclassify the example, this distance – it'll be the minus of the distance, reaching the point, reaching the training example. And you're separating hyperplane. Where you're separating hyperplane is defined by the equation w transpose x + b = 0.

So – oh, well, and I guess also defined these things as the functional margin, geometric margins, respect to training set I defined as the worst case or the minimum functional geometric margin.

So in our development of the optimal margin classifier, our learning algorithm would choose parameters w and b so as to maximize the geometric margin. So our goal is to find the separating hyperplane that separates the positive and negative examples with as large a distance as possible between hyperplane and the positive and negative examples.

And if you go to choose parameters w and b to maximize this, [inaudible] one copy of the geometric margin is that you can actually scale w and b arbitrarily. So you look at this definition for the geometric margin. I can choose to multiply my parameters w and b by 2 or by 10 or any other constant. And it doesn't change my geometric margin.

And one way of interpreting that is you're looking at just separating hyperplane. You look at this line you're separating by positive and negative training examples. If I scale w and b, that doesn't change the position of this plane, though because the equation wh + b = 0 is the same as equation 2 w transpose x + 2b = 0. So it use the same straight line.

And what that means is that I can actually choose whatever scaling for w and b is convenient for me. And in particular, we use in a minute, I can [inaudible] perfect constraint like that the normal w [inaudible] 1 because this means that you can find a solution to w and b. And then by rescaling the parameters, you can easily meet this condition, this rescaled w [inaudible] 1. And so I can add the condition like this and then essentially not change the problem.

Or I can add other conditions. I can actually add a condition that – excuse me, the absolute value of w1 = 1. I can have only one of these conditions right now [inaudible]. And adding condition to the absolute value – the first component of w must be to 1. And again, you can find the absolute solution and just rescale w and meet this condition.

And it can have other, most esoteric conditions like that because again, this is a condition that you can solve for the optimal margin, and then just by scaling, you have w up and down. You can – you can then ensure you meet this condition as well.

So again, [inaudible] one of these conditions right now, not all of them. And so our ability to choose any scaling condition on w that's convenient to us will be useful again in a second.

All right. So let's go ahead and break down the optimization problem. And again, my goal is to choose parameters w and b so as to maximize the geometric margin.

Here's my first attempt at writing down the optimization problem. Actually wrote this one down right at the end of the previous lecture. Begin to solve the parameters gamma w and b such that – that [inaudible]

i for in training examples.

Let's say I choose to add this normalization condition. So the norm condition that w − the normal w is equal to 1 just makes the geometric and the functional margin the same. And so I'm saying I want to find a value − I want to find a value for gamma as big as possible so that all of my training examples have functional margin greater than or equals gamma, and with the constraint that normal w equals 1, functional margin and geometric margin are the same. So it's the same. Find the value for gamma so that all the values − all the geometric margins are greater or equal to gamma.

So you solve this optimization problem, then you have derived the optimal margin classifier − that there's not a very nice optimization problem because this is a nasty, nonconvex constraints. And [inaudible] is asking that you solve for parameters w that lie on the surface of a unisphere, lie on his [inaudible]. It lies on a unicircle − a unisphere.

And so if we can come up with a convex optimization problem, then we'd be guaranteed that our [inaudible] descend to other local [inaudible] will not have local optimal. And it turns out this is an example of a nonconvex constraint. This is a nasty constraint that I would like to get rid of.

So let's change the optimization problem one more time. Now, let me pose a slightly different optimization problem. Let me maximize the functional margin divided by the normal w subject to yi w transpose xi.

So in other words, once you find a number, gamma hat, so that every one of my training examples has functional margin greater than the gamma hat, and my optimization objective is I want to maximize gamma hat divided by the normal w. And so I wanna maximize the function margin divided by the normal w.

And we saw previously the function margin divided by the normal w is just a geometric margin, and so this is a different way of posing the same optimization problem.

[Inaudible] confused, though. Are there questions about this?

Student:[Inaudible] the second statement has to be made of the functional margin y divided by − why don't you just have it the geometric margin? Why do you [inaudible]?

Instructor (Andrew Ng):[Inaudible] say it again?

Student: For the second statement, where we're saying the data of the functional margin is divided [inaudible].

Instructor (Andrew Ng):Oh, I see, yes.

Student:[Inaudible] is that [inaudible]?

Instructor (Andrew Ng):So let's see, this is the function margin, right? This is not the geometric margin.

Student: Yeah.

Instructor (Andrew Ng):So − oh, I want to divide by the normal w of my optimization objective.

Student:I'm just wondering how come you end up dividing also under the second stage [inaudible] the functional margin. Why are you dividing there by the normal w?

Instructor (Andrew Ng):Let's see. I'm not sure I get the question. Let me try saying this again. So here's my goal. My − I want [inaudible]. So let's see, the parameters of this optimization problem where gamma hat w and b − so the convex optimization software solves this problem for some set of parameters gamma w and b. And I'm imposing the constraint that whatever values it comes up with, yi x [inaudible] x5 + b must be greater than gamma hat. And so this means that the functional margin of every example had better be greater than equal to gamma hat. So there's a constraint to the function margin and a constraint to the gamma hat.

But what I care about is not really maximizing the functional margin. What I really care about − in other words, in optimization objective, is maximizing gamma hat divided by the normal w, which is the geometric margin.

So in other words, my optimization [inaudible] is I want to maximize the function margin divided by the normal w. Subject to that, every example must have function margin and at least gamma hat. Does that make sense now?

Student:[Inaudible] when you said that to maximize gamma or gamma hat, respect to gamma w and with respect to gamma hat so that [inaudible] gamma hat are no longer [inaudible]?

Instructor (Andrew Ng): So this is the − so it turns out − so this is how I write down the − this is how I write down an optimization problem in order to solve for the geometric margin. What is it − so it turns out that the question of this − is the gamma hat the function of w and b?

And it turns out that in my previous mathematical definition, it was, but the way I'm going to pose this as an optimization problem is I'm going to ask the convex optimization solvers − and this [inaudible] software − unless you have software for solving convex optimization problems − hen I'm going to pretend that these are independent variables and ask my convex optimization software to find me values for gamma, w, and b, to make this value as big as possible and subject to this constraint.

And it'll turn out that when it does that, it will choose − or obviously, it will choose for gamma to be as big as possible because optimization objective is this: You're trying to maximize gamma hat.

So for x value of w and b, my software, which choose to make gamma hat as big as possible − well, but how big can we make gamma hat? Well, it's limited by use constraints. It says that every training example must have function margin greater than equal to gamma hat. And so my − the bigger you can make gamma hat will be the value of the smallest functional margin. And so when you solve this optimization problem, the value of gamma hat you get out will be, indeed, the minimum of the functional margins of your training set.

Okay, so Justin?

Student: Yeah, I was just wondering, I guess I'm a little confused because it's like, okay, you have two class of data. And you can say, "Okay, please draw me a line such that you maximize the distance between − the smallest distance that [inaudible] between the line and the data points." And it seems like that's kind of what we're doing, but it's − it seems like this is more complicated than that. And I guess I'm wondering what is the difference.

Instructor (Andrew Ng):I see. So I mean, this is − the question is [inaudible]. Two class of data − trying to find separate hyperplane. And this seems more complicated than trying to find a line [inaudible]. So I'm just repeating the questions in case − since I'm not sure how all the audio catches it.

So the answer is this is actually exactly that problem. This is exactly that problem of given the two class of data, positive and negative examples, this is exactly the formalization of the problem where I go is to find a line that separates the two − the positive and negative examples, maximizing the worst-case distance between the [inaudible] point and this line.

Okay? Yeah, [Inaudible]?

Student: So why do you care about the worst-case distance [inaudible]?

Instructor (Andrew Ng): Yeah, let me − for now, why do we care about the worst-case distance? For now, let's just say − let's just care about the worst-case distance for now. We'll come back, and we'll fix that later. We'll − that's a − caring about the worst case is is just − is just a nice way to formulate this optimization problem. I'll come back, and I'll change that later.

Okay, raise your hand if this makes sense − if this formulation makes sense? Okay, yeah, cool.

Great. So let's see − so this is just a different way of posing the same optimization problem. And on the one hand, I've got to get rid of this nasty, nonconvex constraint, while on the other hand, I've now added a nasty, nonconvex objective. In particular, this is not a convex function in parameters w.

And so you can't − you don't have the usual guarantees like if you [inaudible] global minimum. At least that does not follow immediately from this because this is nonconvex.

So what I'm going to do is, earlier, I said that can pose any of a number of even fairly bizarre scaling constraints on w. So you can choose any scaling constraint like this, and things are still fine. And so here's the scaling I'm going to choose to add. Again, I'm gonna assume for the purposes of today's lecture, I'm gonna assume that these examples are linearly separable, that you can actually separate the positive and negative classes, and that we'll come back and fix this later as well.

But here's the scaling constraint I want to impose on w. I want to impose a constraint that the functional margin is equal to 1. And another way of writing that is that I want to impose a constraint that min over i, yi − that in the worst case, function y is over 1.

And clearly, this is a scaling constraint because if you solve for w and b, and you find that your worst-case function margin is actually 10 or whatever, then by dividing through w and b by a factor of 10, I can get

my functional margin to be over 1. So this is a scaling constraint [inaudible] would imply. And this is just more compactly written as follows. This is imposing a constraint that the functional margin be equal to 1.

And so if we just take what I wrote down as No. 2 of our previous optimization problem and add the scaling constraint, we then get the following optimization problem: min over wb.

I guess previously, we had a maximization over gamma hats divided by the normal w. So those maximize 1 over the normal w, but so that's the same as minimizing the normal w squared. It was great. Maximum normal w is min w − normal w squared. And then these are our constraints. Since I've added the constraint, the functional margin is over 1.

And this is actually my final − well, final formulation of the optimal margin classifier problem, at least for now.

So the picture to keep in mind for this, I guess, is that our optimization objective is once you minimize the normal w. And so our optimization objective is just the [inaudible] quadratic function. And [inaudible] those pictures [inaudible] can draw it. So it − if [inaudible] is w1 and w2, and you want to minimize the quadratic function like this − so quadratic function just has [inaudible] that look like this.

And moreover, you have a number of linear constraints in your parameters, so you may have linear constraints that eliminates that half space or linear constraint eliminates that half space [inaudible]. So there's that half space and so on.

And so the picture is you have a quadratic function, and you're ruling out various half spaces where each of these linear constraints. And I hope − if you can picture this in 3D, I guess [inaudible] kinda draw our own 3D, hope you can convince yourself that this is a convex problem that has no local optimum. But they be run great [inaudible] within this set of points that hasn't ruled out, then you convert to the global optimum.

And so that's the convex optimization problem. The − does this [inaudible] nice and [inaudible]. Questions about this?

Actually, just raise your hand if this makes sense. Okay, cool.

So this gives you the optimal margin classifier algorithm. And it turns out that this is the convex optimization problem, so you can actually take this formulation of the problem and throw it at off-the-shelf software − what's called a QP or quadratic program software.

This [inaudible] optimization is called a quadratic program, where the quadratic convex objective function and [inaudible] constraints − so you can actually download software to solve these optimization problems for you. Usually, as you wanna use the − use [inaudible] because you have constraints like these, although you could actually modify [inaudible] work with this, too.

So we could just declare success and say that we're done with this formulation of the problem. But what I'm going to do now is take a digression to talk about primal and duo optimization problems. And in particular, I'm going to − later, I'm going to come back and derive yet another very different form of this optimization problem. And the reason we'll do that is because it turns out this optimization problem has certain properties that make it amenable to very efficient algorithms.

And moreover, I'll be deriving what's called the duo formulation of this that allows us to apply the optimal margin classifier even in very high-dimensional feature spaces − even in sometimes infinite dimensional feature spaces.

So we can come back to that later. But let me know, since I'm talking about convex optimization. So how many here is − how many of you, from, I don't know, calculus, remember the method of Lagrange multipliers for solving an optimization problem like minimum − minimization, maximization problem subject to some constraint? How many of you remember the method of Lagrange multipliers for that?

Oh, okay, cool. Some of you, yeah. So if you don't remember, don't worry. I − I'll describe that briefly here as well, but what I'm really gonna do is talk about the generalization of this method of Lagrange multipliers that you may or may not have seen in some calculus classes. But if you haven't seen it before, don't worry about it.

So the method of Lagrange multipliers is − was − well, suppose there's some function you want to minimize, or minimize f of w. We're subject to some set of constraints that each i of w must equal 0 − for i = 1 [inaudible] l.

And given this constraint, I'll actually usually write it in vectorial form in which I write h of w as this

vector value function. So that is equal to 0, where 0 is the arrow on top. I used that to denote the vector of all 0s.

So you want to solve this optimization problem. Some of you have seen method of Lagrange multipliers where you construct this [inaudible] Lagrange, which is the original optimization objective plus some [inaudible] Lagrange multipliers the highest constraints.

And these parameters – they derive – we call the Lagrange multipliers. And so the way you actually solve the optimization problem is you take the partial derivative of this with respect to the original parameters and set that to 0. So the partial derivative with respect to your Lagrange multipliers [inaudible], and set that to 0. And then the same as theorem through [inaudible], I guess [inaudible] Lagrange was that for w – for some value w star to get a solution, it is necessary that – can this be the star?

Student: Right.

Instructor (Andrew Ng): The backwards e – there exists. So there exists beta star such that those partial derivatives are equal to 0. So the method of Lagrange multipliers is to solve this problem, you construct a Lagrange, take the derivative with respect to the original parameters b, the original parameters w, and with respect to the Lagrange multipliers beta. Set the partial derivatives equal to 0, and solve for our solutions. And then you check each of the solutions to see if it is indeed a minimum.

Great. So great – so what I'm going to do is actually write down the generalization of this. And if you haven't seen that before, don't worry about it. This is [inaudible].

So what I'm going to do is actually write down the generalization of this to solve a slightly more difficult type of constraint optimization problem, which is suppose you want to minimize f of w subject to the constraint that gi of w, excuse me, is less than equal to 0, and that hi of w is equal to 0.

And again, using my vector notation, I'll write this as g of w is equal to 0. And h of w is equal to 0. So in [inaudible]'s case, we now have inequality for constraint as well as equality constraint.

I then have a Lagrange, or it's actually still – called – say generalized Lagrange, which is now a function of my original optimization for parameters w, as well as two sets of Lagrange multipliers, alpha and beta. And so this will be f of w.

Now, here's a cool part. I'm going to define theta subscript p of w to be equal to max of alpha beta subject to the constraints that the alphas are, beta equal to 0 of the Lagrange.

And so I want you to consider the optimization problem min over w of max over alpha beta, such that the alpha is a greater than 0 of the Lagrange. And that's just equal to min over w, theta p of w.

And just to give us a name, the [inaudible] – the subscript p here is a sense of primal problem. And that refers to this entire thing. This optimization problem that written down is called a primal problem. This means there's the original optimization problem in which [inaudible] solving. And later on, I'll derive in another version of this, but that's what p stands for. It's a – this is a primal problem.

Now, I want you to look at – consider theta over p again. And in particular, I wanna consider the problem of what happens if you minimize w – minimize as a function of w this quantity theta over p. So let's look at what theta p of w is. Notice that if gi of w is greater than 0, so let's pick the value of w. And let's ask what is the state of p of w? So if w violates one of your primal problems constraints, then state of p of w would be infinity. Why is that?

[Inaudible] p [inaudible] second. Suppose I pick a value of w that violates one of these constraints. So gi of w is positive. Then – well, theta p is this – maximize this function of alpha and beta – the Lagrange. So one of these gi of w's is this positive, then by setting the other responding alpha i to plus infinity, I can make this arbitrarily large. And so if w violates one of my primal problem's constraints in one of the gis, then max over alpha of this Lagrange will be plus infinity.

There's some of – and in the same way – I guess in a similar way, if hi of w is not equal to 0, then theta p of w also be infinity for a very similar reason because if hi of w is not equal to 0 for some value of i, then in my Lagrange, I had a beta i x hi theorem. And so by setting beta i to be plus infinity or minus infinity depending on the sign of hi, I can make this plus infinity as well.

And otherwise, theta p of w is just equal to f of w. Turns out if I had a value of w that satisfies all of the gi and the hi constraints, then we maximize in terms of alpha and beta – all the Lagrange multiply theorems will actually be obtained by setting all the Lagrange multiply theorems to be 0, and so theta p just left with

f of w.

Thus, theta p of w is equal to f of w if constraints are satisfied [inaudible] the gi in hi constraints, and is equal to plus infinity otherwise.

So the problem I wrote down that minimizes the function of w − theta p of w − this is [inaudible] problem. That's just exactly the same problem as my original primal problem because if you choose a value of w that violates the constraints, you get infinity. And if you satisfy the constraints, you get f of w. So this is really just the same as − well, we'll say, "Satisfy the constraints, and minimize f of w." That's really what minimizing the state of p of w is.

Raise your hand if this makes sense. Yeah, okay, cool. So all right. I hope no one's getting mad at me because I'm doing so much work, and when we come back, it'll be exactly the same thing we started with.

So here's the cool part. Let me know if you find it in your problem. To find theta d and d [inaudible] duo, and this is how the function of alpha and beta is. It's not the function of the Lagrange multipliers. It's not of w. To find this, we minimize over w of my generalized Lagrange.

And my duo problem is this. So in other words, this is max over that. And so this is my duo optimization problem. To maximize over alpha and beta, theta d over alpha and beta. So this optimization problem, I guess, is my duo problem.

I want you to compare this to our previous prime optimization problem. The only difference is that I took the max and min, and I switched the order around with the max and min. That's the difference in the primal and the duo optimization [inaudible].

And it turns out that it's a − it's sort of − it's a fact − it's true, generally, that d star is less than [inaudible] p star. In other words, I think I defined p star previously. P star was a value of the prime optimization problem. And in other words, that it's just generally true that the max of the min of something is less than equal to the min of the max of something. And this is a general fact.

And just as a concrete example, the max over y in the set 01 x − oh, excuse me, of the min of the set in 01 of indicator x = y − this is [inaudible] equal to min.

So this equality − this inequality actually holds true for any function you might find in here. And this is one specific example where the min over xy − excuse me, min over x of [inaudible] equals y − this is always equal to 0 because whatever y is, you can choose x to be something different. So this is always 0, whereas if I exchange the order to min and max, then thing here is always equal to 1. So 0 [inaudible] to 1.

And more generally, this min/max − excuse me, this max/min, thus with the min/max holds true for any function you might put in there.

But it turns out that sometimes under certain conditions, these two optimization problems have the same value. Sometimes under certain conditions, the primal and the duo problems have the same value. And so you might be able to solve the duo problem rather than the primal problem.

And the reason to do that is that sometimes, which we'll see in the optimal margin classifier problem, the support vector machine problem, the duo problem turns out to be much easier than it − often has many useful properties that will make user compared to the primal.

So for the sake of − so what I'm going to do now is write down formally the certain conditions under which that's true − where the primal and the duo problems are equivalent. And so our strategy for working out the [inaudible] of support vector machine algorithm will be that we'll write down the primal optimization problem, which we did previously, and maximizing classifier.

And then we'll derive the duo optimization problem for that. And then we'll solve the duo problem. And by modifying that a little bit, that's how we'll derive this support vector machine.

But let me ask you − for now, let me just first, for the sake of completeness, I just write down the conditions under which the primal and the duo optimization problems give you the same solutions. So let f be convex. If you're not sure what convex means, for the purposes of this class, you can take it to mean that the Hessian, h is positive. [Inaudible], so it just means it's a [inaudible] function like that.

And once you learn more about optimization − again, please come to this week's discussion session taught by the TAs.

Then suppose hi − the hi constraints [inaudible], and what that means is that hi of w equals alpha i transpose w plus vi. This actually means the same thing as linear. Without the term b here, we say that hi

is linear where we have a constant interceptor as well. This is technically called [inaudible] other than linear.

And let's suppose that gi's are strictly feasible. And what that means is that there is just a value of the w such that from i, gi of w is less than 0. Don't worry too much [inaudible]. I'm writing these things down for the sake of completeness, but don't worry too much about all the technical details. Strictly feasible, which just means that there's a value of w such that all of these constraints are satisfy were stricter than the equality rather than what less than equal to.

Under these conditions, there were exists w star, alpha star, beta star such that w star solves the primal problem. And alpha star and beta star, the Lagrange multipliers, solve the duo problem. And the value of the primal problem will be equal to the value of the duo problem will be equal to the value of your Lagrange multiplier – excuse me, will be equal to the value of your generalized Lagrange, the value of that w star, alpha star, beta star.

In other words, you can solve either the primal or the duo problem. You get the same solution.

Further, your parameters will satisfy these conditions. Partial derivative perspective parameters would be 0. And actually, to keep this equation in mind, we'll actually use this in a second when we take the Lagrange, and we – and our support vector machine problem, and take a derivative with respect to w to solve a – to solve our – to derive our duo problem. We'll actually perform this step ourselves in a second.

Partial derivative with respect to the Lagrange multiplier beta is equal to 0. Turns out this will hold true, too. This is called the – well – this is called the KKT complementary condition. KKT stands for Karush-Kuhn-Tucker, which were the authors of this theorem. Well, and by tradition, usually this [inaudible] KKT conditions. But the other two are – just so the [inaudible] is greater than 0, which we had previously and that your constraints are actually satisfied.

So let's see. [Inaudible] All right. So let's take those and apply this to our optimal margin optimization problem that we had previously. I was gonna say one word about this, which is – was gonna say one word about this KTT complementary condition is that a condition that is a – at your solution, you must have that alpha star i times gi of w is equal to 0.

So let's see. So the product of two numbers is equal to 0. That means that at least one of these things must be equal to 0. For the product of two things to be equal to 0, well, that's just saying either alpha or i or gi is equal to 0.

So what that implies is that the – just Karush-Kuhn-Tucker – most people just say KKT, but we wanna show you the right spelling of their names. So KKT complementary condition implies that if alpha i is not 0, that necessarily implies that gi of w star is equal to 0. And usually, it turns out – so all the KKT condition guarantees is that at least one of them is 0. It may actually be the case that both alpha and gi are both equal to 0. But in practice, when you solve this optimization problem, you find that to a large part, alpha i star is not equal to 0 if and only gi of w star 0, 0.

This is not strictly true because it's possible that both of these may be 0. But in practice, when we – because when we solve problems like these, you're, for the most part, usually exactly one of these will be non-0.

And also, when this holds true, when gi of w star is equal to 0, we say that gi – gi of w, I guess, is an active constraint because we call a constraint – our constraint was a gi of w must be less than or equal to 0. And so it is equal to 0, then we say that that's a constraint that this is an active constraint.

Once we talk about [inaudible], we come back and [inaudible] and just extend this idea a little bit more. [Inaudible] board. [Inaudible] turn to this board in a second, but – so let's go back and work out one of the primal and the duo optimization problems for our optimal margin classifier for the optimization problem that we worked on just now.

As a point of notation, in whatever I've been writing down so far in deriving the KKT conditions, when Lagrange multipliers were alpha i and beta i, it turns out that when applied as [inaudible] dm, turns out we only have one set of Lagrange multipliers alpha i.

And also, as I was working out the KKT conditions, I used w to denote the parameters of my primal optimization problem. [Inaudible] I wanted to minimize f of w. In my very first optimization problem, I had that optimization problem [inaudible] finding the parameters w.

In my svn problem, I'm actually gonna have two sets of parameters, w and b. So this is just a – keep

that sort of slight notation change in mind.

So problem we worked out previously was we want to minimize the normal w squared and just add a half there by convention because it makes other work − math work a little nicer. And subject to the constraint that yi x w [inaudible] xi + v must be = greater than 1.

And so let me just take this constraint, and I'll rewrite it as a constraint. It's gi of w, b. Again, previously, I had gi of w, but now I have parameters w and b. So gi of w, b defined as 1.

So let's look at the implications of this in terms of the KKT duo complementary condition again. So we have that alpha i is basically equal to 0. That necessarily implies that gi of w, b is equal to 0. In other words, this is an active constraint.

And what does this mean? It means that it actually turns out gi of wb equal to 0 that is − that means exactly that the training example xi, yi has functional margin equal to 1. Because this constraint was that the functional margin of every example has to be greater equal to 1. And so if this is an active constraint, it just − inequality holds that equality. That means that my training example i must have functional margin equal to exactly 1.

And so − actually, yeah, right now, I'll do this on a different board, I guess. So in pictures, what that means is that, you have some training sets, and you'll have some separating hyperplane. And so the examples with functional margin equal to 1 will be exactly those which are − so they're closest to my separating hyperplane.

So that's my equation. [Inaudible] equal to 0. And so in this − in this cartoon example that I've done, it'll be exactly − these three examples that have functional margin equal to 1, and all of the other examples as being further away than these three will have functional margin that is strictly greater than 1.

And the examples with functional margin equal to 1 will usually correspond to the ones where the corresponding Lagrange multipliers also not equal to 0. And again, it may not hold true. It may be the case that gi and alpha i equal to 0. But usually, when gi's not − is 0, alpha i will be non-0.

And so the examples of functional margin equal to 1 will be the ones where alpha i is not equal to 0.

One useful property of this is that as suggested by this picture and so true in general as well, it turns out that we find a solution to this − to the optimization problem, you find that relatively few training examples have functional margin equal to 1.

In this picture I've drawn, there are three examples with functional margin equal to 1. There are just few examples of this minimum possible distance to your separating hyperplane. And these are three − these examples of functional margin equal to 1 − they are what we're going to call the support vectors. And this needs the name support vector machine. There'll be these three points with functional margin 1 that we're calling support vectors.

And the fact that they're relatively few support vectors also means that usually, most of the alpha i's are equal to 0. So with alpha i equal to 0, for examples, though, not support vectors.

Let's go ahead and work out the actual optimization problem. So we have a [inaudible] margin optimization problem. So there we go and write down the margin, and because we only have inequality constraints where we really have gi star constraints, no hi star constraint. We have inequality constraints and no equality constraints, I'll only have Lagrange multipliers of type alpha − no betas in my generalized Lagrange. But my Lagrange will be one-half w squared minus. That's my Lagrange.

And so let's work out what the duo problem is. And to do that, I need to figure out what theta d of alpha − and I know again, beta's there − so what theta d of alpha is min with respect to wb of lb alpha. So the duo problem is the maximize theta d as the function of alpha. So as to work out what theta d is, and then that'll give us our duo problem.

So then to work out what this is, what do you need to do? We need to take a look at Lagrange and minimize it as a function of lv and b so − and what is this?

How do you minimize Lagrange? So in order to minimize the Lagrange as a function of w and b, we do the usual thing. We take the derivatives of w − Lagrange with respect to w and b. And we set that to 0. That's how we minimize the Lagrange with respect to w and b.

So take the derivative with respect to w of the Lagrange. And I want − I just write down the answer. You know how to do calculus like this. So I wanna minimize this function of w, so I take the derivative and set it to 0. And I get that. And then so this implies that w must be that.

And so w, therefore, is actually a linear combination of your input feature vectors xi. This is sum of your various weights given by the alpha i's and times the xi's, which are your examples in your training set. And this will be useful later.

The other equation we have is – here, partial derivative of Lagrange with respect to p is equal to minus sum of i plus 1 to m [inaudible] for i. And so I'll just set that to equal to 0. And so these are my two constraints. And so [inaudible].

So what I'm going to do is I'm actually going to take these two constraints, and well, I'm going to take whatever I thought to be the value for w. And I'm gonna take what I've worked out to be the value for w, and I'll plug it back in there to figure out what the Lagrange really is when I minimize with respect to w. [Inaudible] and I'll deal with b in a second.

So let's see. So my Lagrange is 1/2 w transpose w minus. So this first term, w transpose w – this becomes sum y equals one to m, alpha i, yi, xi transpose. This is just putting in the value for w that I worked out previously.

But since this is w transpose w – and so when they expand out of this quadratic function, and when I plug in w over there as well, I find that I have that. Oh, where I'm using these angle brackets to denote end product, so this thing here, it just means the end product, xi transpose xj. And the first and second terms are actually the same except for the minus one half. So to simplify to be equal to that.

So let me go ahead and call this w of alpha. My duo problem is, therefore, the following. I want to maximize w of alpha, which is that [inaudible]. And I want to the – I realize the notation is somewhat unfortunate. I'm using capital W of alpha to denote that formula I wrote down earlier.

And then we also had our lowercase w. The original [inaudible] is the primal problem. Lowercase w transpose xi. So uppercase and lowercase w are totally different things, so unfortunately, the notation is standard as well, as far as I know, so. So the duo problem is that subject to the alpha [inaudible] related to 0, and we also have that the sum of i, yi, alpha i is related to 0.

That last constraint was the constraint I got from this – the sum of i – sum of i, yi alpha i equals to 0. But that's where that [inaudible] came from.

Let me just – I think in previous years that I taught this, where this constraint comes from is just – is slightly confusing. So let me just take two minutes to say what the real interpretation of that is. And if you don't understand it, it's not a big deal, I guess.

So when we took the partial derivative of the Lagrange with respect to b, we end up with this constraint that sum of i, yi, alpha i must be equal to 0. The interpretation of that, it turns out, is that if sum of i, yi, alpha i is not equal to 0, then theta d of wb is – actually, excuse me. Then theta d of alpha is equal to minus infinity for minimizing.

So in other words, it turns out my Lagrange is actually a linear function of my parameters b. And so the interpretation of that constraint we worked out previously was that if sum of i or yi, alpha i is not equal to 0, then theta d of alpha is equal to minus infinity. And so if your goal is to maximize as a function of alpha, theta d of alpha, then you've gotta choose values of alpha for which sum of yi alpha is equal to 0.

And then when sum of yi alpha is equal to 0, then theta d of alpha is equal to w of alpha. And so that's why we ended up deciding to maximize w of alpha subject to that sum of yi alpha is equal to 0.

Yeah, the – unfortunately, the fact of that d would be [inaudible] adds just a little bit of extra notation in our derivation of the duo. But by the way, and [inaudible] all the action of the optimization problem is with w because b is just one parameter.

So let's check. Are there any questions about this?

Okay, cool. So what derived a duo optimization problem – and really, don't worry about this if you're not quite sure where this was. Just think of this as we worked out this constraint, and we worked out, and we took partial derivative with respect to b, that this constraint has the [inaudible] and so I just copied that over here.

But so – worked out the duo of the optimization problem, so our approach to finding – to deriving the optimal margin classifier or support vector machine will be that we'll solve along this duo optimization problem for the parameters alpha star. And then if you want, you can then – this is the equation that we worked out on the previous board. We said that w – this [inaudible] alpha – w must be equal to that. And so

once you solve for alpha, you can then go back and quickly derive w in parameters to your primal problem. And we worked this out earlier.

And moreover, once you solve alpha and w, you can then focus back into your – once you solve for alpha and w, it's really easy to solve for v, so that b gives us the interpretation of [inaudible] training set, and you found the direction for w. So you know where your separating hyperplane's direction is. You know it's got to be one of these things. And you know the orientation and separating hyperplane. You just have to decide where to place this hyperplane. And that's what solving b is.

So once you solve for alpha and w, it's really easy to solve b. You can plug alpha and w back into the primal optimization problem and solve for b.

And I just wrote it down for the sake of completeness, but – and the intuition behind this formula is just that find the worst positive [inaudible] and the worst negative example. Let's say this one and this one – say [inaudible] and [inaudible] the difference between them. And that tells you where you should set the threshold for where to place the separating hyperplane.

And then that's the – this is the optimal margin classifier. This is also called a support vector machine. If you do not use one y [inaudible], it's called kernels. And I'll say a few words about that.

But I hope the process is clear. It's a duo problem. We're going to solve the duo problem for the alpha i's. That gives us w, and that gives us b.

So there's just one more thing I wanna point out as I lead into the next lecture, which is that – I'll just write this out again, I guess – which is that it turns out we can take the entire algorithm, and we can express the entire algorithm in terms of inner products. And here's what I mean by that.

So say that the parameters w is the sum of your input examples. And so we need to make a prediction. Someone gives you a new value of x. You want a value of the hypothesis on the value of x. That's given by g of w transpose x plus b, or where g was this threshold function that outputs minus 1 or plus 1. And so you need to compute w transpose x plus b. And that is equal to alpha i, yi.

And that can be expressed as a sum of these inner products between your training examples and this new value of x [inaudible] value [inaudible].

And this will lead into our next lecture, which is the idea of kernels. And it turns out that in the source of feature spaces where used to support vector machines – it turns out that sometimes your training examples may be very high-dimensional. It may even be the case that the features that you want to use are inner-dimensional feature vectors.

But despite this, it'll turn out that there'll be an interesting representation that you can use that will allow you to compute inner products like these efficiently. And this holds true only for certain feature spaces. It doesn't hold true for arbitrary sets of features.

But we talk about the idea of kernels. In the next lecture, we'll see examples where even though you have extremely high-dimensional feature vectors, you can compute – you may never want to represent xi, x plus [inaudible] inner-dimensional feature vector. You can even store in computer memory. But you will nonetheless be able to compute inner products between different [inaudible] feature vectors very efficiently. And so you can – for example, you can make predictions by making use of these inner products.

This is just xi transpose. You will compute these inner products very efficiently and, therefore, make predictions. And this pointed also – the other reason we derive the duo was because on this board, when we worked out what w of alpha is, w of alpha – actually are the same property – w of alpha is again written in terms of these inner products.

And so if you actually look at the duo optimization problem and step – for all the steps of the algorithm, you'll find that you actually do everything you want – learn the parameters of alpha. So suppose you do an optimization problem, go into parameters alpha, and you do everything you want without ever needing to represent xi directly. And all you need to do is represent this compute inner products with your feature vectors like these.

Well, one last property of this algorithm that's kinda nice is that I said previously that the alpha i's are 0 only for the – are non-0 only for the support vectors, only for the vectors that function y [inaudible] 1.

And in practice, there are usually fairly few of them. And so what this means is that if you're representing w this way, then w when represented as a fairly small fraction of training examples because mostly alpha

i's is 0 − and so when you're summing up the sum, you need to compute inner products only if the support vectors, which is usually a small fraction of your training set. So that's another nice [inaudible] because [inaudible] alpha is 0.

And well, much of this will make much more sense when we talk about kernels.

[Inaudible] quick questions before I close? Yeah.

Student: It seems that for anything we've done the work, the point file has to be really well behaved, and if any of the points are kinda on the wrong side −

Instructor (Andrew Ng): No, oh, yeah, so again, for today's lecture asks you that the data is linearly separable − that you can actually get perfect [inaudible]. I'll fix this in the next lecture as well. But excellent assumption.

Yes?

Student: So can't we assume that [inaudible] point [inaudible], so [inaudible] have [inaudible]?

Instructor (Andrew Ng): Yes, so unless I − says that there are ways to generalize this in multiple classes that I probably won't [inaudible] − but yeah, that's generalization [inaudible].

Okay. Let's close for today, then. We'll talk about kernels in our next lecture.

[End of Audio]

Duration: 77 minutes

1.8 Machine Learning Lecture 8[8]

Figure 1.8

MachineLearning-Lecture08

Instructor (Andrew Ng): Okay. Good morning. Welcome back. If you haven't given me the homework yet, you can just give it to me at the end of class. That's fine. Let's see. And also just a quick reminder − I've actually seen project proposals start to trickle in already, which is great. As a reminder, project proposals are due this Friday, and if any of you want to meet and chat more about project ideas, I also have office hours immediately after lecture today. Are there any questions about any of that before I get started today? Great.

Okay. Welcome back. What I want to do today is wrap up our discussion on support vector machines and in particular we'll also talk about the idea of kernels and then talk about [inaudible] and then I'll talk about the SMO algorithm, which is an algorithm for solving the optimization problem that we posed last time.

To recap, we wrote down the following context optimization problem. All this is assuming that the data is linearly separable, which is an assumption that I'll fix later, and so with this optimization problem, given a training set, this will find the optimal margin classifier for the data set that maximizes this geometric margin from your training examples.

And so in the previous lecture, we also derived the dual of this problem, which was to maximize this. And this is the dual of our primal [inaudible] optimization problem. Here, I'm using these angle brackets to denote inner product, so this is just XI transpose XJ for vectors XI and XJ. We also worked out the ways W would be given by sum over I alpha I YI XI.

Therefore, when you need to make a prediction of classification time, you need to compute the value of the hypothesis applied to an [inaudible], which is G of W transpose X plus B where G is that threshold

---

[8]This content is available online at <http://cnx.org/content/m45905/1.2/>.

function that outputs plus one and minus one. And so this is G of sum over I alpha I. So that can also be written in terms of inner products between input vectors X.

So what I want to do is now talk about the idea of kernels, which will make use of this property because it turns out you can take the only dependers of the algorithm on X is through these inner products. In fact, you can write the entire algorithm without ever explicitly referring to an X vector [inaudible] between input feature vectors. And the idea of a high kernel is as following – let's say that you have an input attribute. Let's just say for now it's a real number. Maybe this is the living area of a house that you're trying to make a prediction on, like whether it will be sold in the next six months.

Quite often, we'll take this feature X and we'll map it to a richer set of features. So for example, we will take X and map it to these four polynomial features, and let me acutely call this mapping Phi. So we'll let Phi of X denote the mapping from your original features to some higher dimensional set of features.

So if you do this and you want to use the features Phi of X, then all you need to do is go back to the learning algorithm and everywhere you see XI, XJ, we'll replace it with the inner product between Phi of XI and Phi of XJ. So this corresponds to running a support vector machine with the features given by Phi of X rather than with your original one-dimensional input feature X.

And in a scenario that I want to consider, sometimes Phi of X will be very high dimensional, and in fact sometimes Phi of X – so for example, Phi of X may contain very high degree polynomial features. Sometimes Phi of X will actually even be an infinite dimensional vector of features, and the question is if Phi of X is an extremely high dimensional, then you can't actually compute to these inner products very efficiently, it seems, because computers need to represent an extremely high dimensional feature vector and then take [inaudible] inefficient.

It turns out that in many important special cases, we can write down – let's call the kernel function, denoted by K, which will be this, which would be inner product between those feature vectors. It turns out there will be important special cases where computing Phi of X is computationally very expensive – maybe is impossible.

There's an infinite dimensional vector, and you can't compute infinite dimensional vectors. There will be important special cases where Phi of X is very expensive to represent because it is so high dimensional, but nonetheless, you can actually compute a kernel between XI and XJ. You can compute the inner product between these two vectors very inexpensively.

And so the idea of the support vector machine is that everywhere in the algorithm that you see these inner products, we're going to replace it with a kernel function that you can compute efficiently, and that lets you work in feature spaces Phi of X even if Phi of X are very high dimensional. Let me now say how that's done. A little bit later today, we'll actually see some concrete examples of Phi of X and of kernels. For now, let's just think about constructing kernels explicitly. This best illustrates my example.

Let's say you have two inputs, X and Z. Normally I should write those as XI and XJ, but I'm just going to write X and Z to save on writing. Let's say my kernel is K of X, Z equals X transpose Z squared. And so this is – right? X transpose Z – this thing here is X transpose Z and this thing is X transpose Z, so this is X transpose Z squared. And that's equal to that. And so this kernel corresponds to the feature mapping where Phi of X is equal to – and I'll write this down for the case of N equals free, I guess.

And so with this definition of Phi of X, you can verify for yourself that this thing becomes the inner product between Phi of X and Phi of Z, because to get an inner product between two vectors is – you can just take a sum of the corresponding elements of the vectors. You multiply them. So if this is Phi of X, then the inner product between Phi of X and Phi of Z will be the sum over all the elements of this vector times the corresponding elements of Phi of Z, and what you get is this one.

And so the cool thing about this is that in order to compute Phi of X, you need [inaudible] just to compute Phi of X. If N is a dimension of X and Z, then Phi of X is a vector of all pairs of XI XJ multiplied of each other, and so the length of Phi of X is N squared. You need order N squared time just to compute Phi of X.

But to compute K – to compute the kernel function, all you need is order N time, because the kernel function is defined as X transpose Z squared, so you just take the inner product between X and Z, which is order N time and you square that and you've computed this kernel function, and so you just computed the

inner product between two vectors where each vector has N squared elements, but you did it in N square time.

Student: For any kernel you find for X and Z, does Phi exist for X and Z?

Instructor (Andrew Ng): Let me talk about that later. We'll talk about what is a valid kernel later. Please raise your hand if this makes sense. So let me just describe a couple of quick generalizations to this. One is that if you define KXZ to be equal to X transpose Z plus C squared, so again, you can compute this kernel in order N time, then that turns out to correspond to a feature vector where I'm just going to add a few more elements at the bottom where you add root 2. Let me read that. That was root 2 CX1 root 2 CX2 root 2 CX3 and C.

And so this is a way of creating a feature vector with both the monomials, meaning the first order terms, as well as the quadratic or the inner product terms between XI and XJ, and the parameter C here allows you to control the relative waiting between the monomial terms, so the first order terms, and the quadratic terms. Again, this is still inner product between vectors of length and square [inaudible] in order N time.

More generally, here are some other examples of kernels. Actually, a generalization of the one I just derived right now would be the following kernel. And so this corresponds to using all N plus DQZ features of all monomials. Monomials just mean the products of XI XJ XK. Just all the polynomial terms up to degree D and plus [inaudible] so on the order of N plus D to the power of D, so this grows exponentially in D.

This is a very high dimensional feature vector, but again, you can implicitly construct the feature vector and take inner products between them. It's very computationally efficient, because you just compute the inner product between X and Z, add C and you take that real number to the power of D and by plugging this in as a kernel, you're implicitly working in an extremely high dimensional computing space.

So what I've given is just a few specific examples of how to create kernels. I want to go over just a few specific examples of kernels. So let's you ask you more generally if you're faced with a new machine-learning problem, how do you come up with a kernel? There are many ways to think about it, but here's one intuition that's sort of useful. So given a set of attributes of X, you're going to use a feature vector of Phi of X and given a set of attributes Z, you're going to use an input feature vector Phi of Z, and so the kernel is computing the inner product between Phi of X and Phi of Z.

And so one intuition – this is a partial intuition. This isn't as rigorous intuition that it is used for. It is that if X and Z are very similar, then Phi of X and Phi of Z will be pointing in the same direction, and therefore the inner product would be large. Whereas in contrast, if X and Z are very dissimilar, then Phi of X and Phi of Z may be pointing different directions, and so the inner product may be small. That intuition is not a rigorous one, but it's sort of a useful one to think about.

If you're faced with a new learning problem – if I give you some random thing to classify and you want to decide how to come up with a kernel, one way is to try to come up with the function P of XZ that is large, if you want to learn the algorithm to think of X and Z as similar and small. Again, this isn't always true, but this is one of several intuitions. So if you're trying to classify some brand new thing – you're trying to classify [inaudible] or DNA sequences or something, some strange thing you want to classify, one thing you could do is try to come up with a kernel that's large when you want the algorithm to think these are similar things or these are dissimilar.

And so this answers the question of let's say I have something I want to classify, and let's say I write down the function that I think is a good measure of how similar or dissimilar X and Z are for my specific problem. Let's say I write down K of XZ equals E to the minus. Let's say I write down this function, and I think this is a good measure of how similar X and Z are. The question, then, is is this really a valid kernel? In other words, to understand how we can construct kernels – if I write down the function like that, the question is does there really exist some Phi such that KXZ is equal to the inner product?

And that's the question of is K a valid kernel. It turns out that there is a result that characterizes necessary and sufficient conditions for when a function K that you might choose is a valid kernel. I should go ahead show part of that result now.

Suppose K is a valid kernel, and when I say K is a kernel, what I mean is there does indeed exist some function Phi for which this holds true. Then let any set of points X1 up to XM be given. Let me define a matrix K. I apologize for overloading notation. K I'm going to use to denote both the kernel function, which

is the function of X and Z as well as a matrix. Unfortunately, there aren't enough alphabets. Well, that's not true.

We need to find the kernel matrix to be an M-by-M matrix such that K subscript IJ is equal to the kernel function applied to two of my examples. Then it turns out that for any vector Z that's indimensional, I want you to consider Z transpose KZ. By definition of matrix multiplication, this is that, and so KIJ is a kernel function between XI and XJ, so that must equal to this. I assume that K is a valid kernel function, and so there must exist such a value for Phi. This is the inner product between two feature vectors, so let me just make that inner product the explicit.

I'm going to sum over the elements of this vector, and I'm going to use Phi XI subscript K just to denote the K element of this vector. Just rearrange sums. You get sum over K. This next set may look familiar to some of you, which is just – right? Therefore, this is the sum of squares and it must therefore be greater than or equal to zero. Do you want to take a minute to look for all the steps and just make sure you buy them all? Oh, this is the inner product between the vector of Phi of XI and Phi of XJ, so the inner product between two vectors is the sum over all the elements of the vectors of the corresponding element.

Student:[Inaudible].

Instructor (Andrew Ng):Oh, yes it is. This is just A transpose B equals sum over K, AK BK, so that's just this. This is the sum of K of the K elements of this vector. Take a look at this and make sure it makes sense. Questions about this? So just to summarize, what we showed was that for any vector Z, Z transpose KZ is greater than or equal to zero, and this is one of the standard definitions of a matrix, the matrix K being posisemidefinite when a matrix K is posisemidefinite, that is, K is equal to zero.

Just to summarize, what was shown is that if K is a valid kernel – in other words, if K is a function for which there exists some Phi such that K of XI XJ is the inner product between Phi of XI and Phi of XJ. So if K is a valid kernel, we showed, then, that the kernel matrix must be posisemidefinite. It turns out that the converse [inaudible] and so this gives you a test for whether a function K is a valid kernel.

So this is a theorem due to Mercer, and so kernels are also sometimes called Mercer kernels. It means the same thing. It just means it's a valid kernel. Let K of XZ be given. Then K is a valid kernel – in other words, it's a Mercer kernel, i.e., there exists a Phi such that KXZ equals Phi of X transpose Phi of Z – if and only if for any set of M examples, and this really means for any set of M points. It's not necessarily a training set. It's just any set of M points you may choose. It holds true that the kernel matrix, capital K that I defined just now, is symmetric posisemidefinite.

And so I proved only one direction of this result. I proved that if it's a valid kernel, then K is symmetry posisemidefinite, but the converse I didn't show. It turns out that this is necessary and a sufficient condition. And so this gives you a useful test for whether any function that you might want to choose is a kernel.

A concrete example of something that's clearly not a valid kernel would be if you find an input X such that K of X, X – and this is minus one, for example – then this is an example of something that's clearly not a valid kernel, because minus one cannot possibly be equal to Phi of X transpose Phi of X, and so this would be one of many examples of functions that will fail to meet the conditions of this theorem, because inner products of a vector itself are always greater than zero.

So just to tie this back explicitly to an SVM, let's say to use a support vector machine with a kernel, what you do is you choose some function K of XZ, and so you can choose – and it turns out that function I wrote down just now – this is, indeed, a valid kernel. It is called the Galcean kernel because of the similarity to Galceans. So you choose some kernel function like this, or you may choose X transpose Z plus C to the D vector.

To apply a support vector machine kernel, you choose one of these functions, and the choice of this would depend on your problem. It depends on what is a good measure of one or two examples similar and one or two examples different for your problem. Then you go back to our formulation of support vector machine, and you have to use the dual formulation, and you then replace everywhere you see these things, you replace it with K of XI, XJ.

And you then run exactly the same support vector machine algorithm, only everywhere you see these inner products, you replace them with that, and what you've just done is you've taken a support vector machine and you've taken each of your feature vectors X and you've replaced it with implicitly a very high

dimensional feature vector.

It turns out that the Galcean kernel corresponds to a feature vector that's infinite dimensional. Nonetheless, you can run a support vector machine in a finite amount of time, even though you're working with infinite dimensional feature vectors, because all you ever need to do is compute these things, and you don't ever need to represent these infinite dimensional feature vectors explicitly.

Why is this a good idea? It turns out – I think I started off talking about support vector machines. I started saying that we wanted to start to develop non-linear learning algorithms. So here's one useful picture to keep in mind, which is that let's say your original data – I didn't mean to draw that slanted. Let's say you have one-dimensional input data. You just have one feature X and R. What a kernel does is the following. It takes your original input data and maps it to a very high dimensional feature space.

In the case of Galcean kernels, an infinite dimensional feature space – for pedagogical reasons, I'll draw two dimensions here. So say [inaudible] very high dimensional feature space where – like so. So it takes all your data in R1 and maps it to R infinity, and then you run a support vector machine in this infinite dimensional space and also exponentially high dimensional space, and you'll find the optimal margin classifier – in other words, the classifier that separates your data in this very high dimensional space with the largest possible geometric margin.

In this example that you just drew anyway, whereas your data was not linearly separable in your originally one dimensional space, when you map it to this much higher dimensional space, it becomes linearly separable, so you can use your linear classifier to [inaudible] which data is not really separable in your original space. This is what support vector machines output nonlinear decision boundaries and in the entire process, all you ever need to do is solve complex optimization problems. Questions about any of this?

Student:[Inaudible] sigmer?

Instructor (Andrew Ng): Yeah, so sigmer is – let's see. Well, I was going to talk about [inaudible] later. One way to choose sigmer is save aside a small amount of your data and try different values of sigmer and train an SVM using, say, two thirds of your data. Try different values of sigmer, then see what works best on a separate hold out cross validation set – on a separate set that you're testing.

Something about learning algorithms we talked about – locally [inaudible] linear aggressions [inaudible] bandwidth parameter, so there are a number of parameters to some of these algorithms that you can choose IDs by saving aside some data to test on. I'll talk more about model selection [inaudible] explicitly. Are there other questions?

Student: So how do you know that moving it up to high dimensional space is going to give you that kind of separation?

Instructor (Andrew Ng): Good question. Usually, you don't know [inaudible]. Sometimes you can know, but in most cases, you won't know [inaudible] actually going to linearly separable, so the next topic will be [inaudible], which is what [inaudible] SVMs that work even though the data is not linearly separable.

Student: If you tend linearly separated by mapping a higher dimension, couldn't you also just use [inaudible] higher dimension?

Instructor (Andrew Ng): So very right. This is a question about what to do if you can't separate it in higher dimensional space. Let me try to address that work with a discussion of [inaudible] soft margin SVMs. Okay.

Student: What if you run an SVM algorithm that assumes the data are linearly separable on data that is not actually linearly separable?

Instructor (Andrew Ng): You guys are really giving me a hard time about whether the data's linearly separable. It turns out this algorithm won't work if the data is not linearly separable, but I'll change that in a second and make it work. If I move on to talk about that, let me just say one final word about kernels, which is that I talked about kernels in a context of support vector machines, and the idea of kernels was what really made support vector machines a very powerful learning algorithm, and actually towards the end of today's lecture if I have time, I'll actually give a couple more [inaudible] examples of how to choose kernels as well.

It turns out that the idea of kernels is actually more general than support vector machines, and in particular, we took this SVM algorithm and we derived a dual, and that was what let us write the entire

algorithm in terms of inner products of these. It turns out that you can take many of the other algorithms that you've seen in this class – in fact, it turns out you can take most of the linear algorithms we talked about, such as linear regression, logistic regression [inaudible] and it turns out you can take all of these algorithms and rewrite them entirely in terms of these inner products.

So if you have any algorithm that you can rewrite in terms of inner products, then that means you can replace it with K of XI XJ and that means that you can take any of theses algorithms and implicitly map the features vectors of these very high dimensional feature spaces and have the algorithm still work.

The idea of kernels is perhaps most widely used with support vector machines, but it is actually more general than that, and you can take many of the other algorithms that you've seen and many of the algorithms that we'll see later this quarter as well and write them in terms of inner products and thereby kernalize them and apply them to infinite dimensional feature spaces. You'll actually get to play with many of these ideas more in the next problem set.

Let's talk about non-linear decision boundaries, and this is the idea of – it's called the L1 norm soft margin SVM. Machine only people sometimes aren't great at coming up with good names, but here's the idea. Let's say I have a data set. This is a linearly separable data set, but what I do if I have a couple of other examples there that makes the data non-linearly separable, and in fact, sometimes even if the data is linearly separable, maybe you might not want to.

So for example, this is a very nice data set. It looks like there's a great decision boundary that separates the two [inaudible]. Well, what if I had just one outlier down here? I could still linearly separate this data set with something like that, but I'm somehow letting one slightly suspicious example skew my entire decision boundary by a lot, and so what I'm going to talk about now is the L1 norm soft margin SVM, which is a slightly modified formulation of the SVM optimization problem.

They will let us deal with both of these cases – one where one of the data's just not linearly separable and two, what if you have some examples that you'd rather not get [inaudible] in a training set. Maybe with an outlier here, maybe you actually prefer to choose that original decision boundary and not try so hard to get that training example. Here's the formulation. Our SVM primal problem was to minimize one-half [inaudible] W squared.

So this is our original problem, and I'm going to modify this by adding the following. In other words, I'm gonna add these penalty terms, CIs, and I'm going to demand that each of my training examples is separated with functional margin greater than or equal to one minus CI, and you remember if this is greater than zero – was it two lectures ago that I said that if the function margin is greater than zero, that implies you classified it correctly. If it's less than zero, then you misclassified it.

By setting some of the CIs to be larger than one, I can actually have some examples with functional margin negative, and therefore I'm allowing my algorithm to misclassify some of the examples of the training set. However, I'll encourage the algorithm not to do that by adding to the optimization objective, this sort of penalty term that penalizes setting CIs to be large. This is an optimization problem where the parameters are WB and all of the CIs and this is also a convex optimization problem. It turns out that similar to how we worked on the dual of the support vector machine, we can also work out the dual for this optimization problem.

I won't actually do it, but just to show you the steps, what you do is you construct [inaudible] Alpha R, and I'm going to use Alpha and R to denote the [inaudible] multipliers no corresponding to this set of constraints that we had previously and this new set of constraints on the CI [inaudible] zero. This gives us a use of the [inaudible] multipliers. The [inaudible] will be optimization objective minus sum from plus CI minus – and so there's our [inaudible] optimization objective minus or plus Alpha times each of these constraints, which are greater or equal to zero.

I won't redivide the entire dual again, but it's really the same, and when you derive the dual of this optimization problem and when you simplify, you find that you get the following. You have to maximize [inaudible], which is actually the same as before. So it turns out when you derive the dual and simply, it turns out that the only way the dual changes compared to the previous one is that rather than the constraint that the Alpha [inaudible] are greater than or equal to zero, we now have a constraint that the Alphas are between zero and C.

This derivation isn't very hard, and you're encouraged to go home and try to do it yourself. It's really essentially the same math, and when you simply, it turns out you can simply the R of the [inaudible] multiplier away and you end up with just these constraints of the Alphas.

Just as an aside, I won't derive these, either. It turns out that – remember, I wrote down the [inaudible] conditions in the last lecture. The necessary conditions for something to be an optimal solution to constrain optimization problems. So if you used the [inaudible] conditions, it turns out you can actually derive conversions conditions, so we want to solve this optimization problem. When do we know the Alphas have converged to the global optimum?

It turns out you can use the following. I don't want to say a lot about these. It turns out from the [inaudible] conditions you can derive these as the conversion conditions for an algorithm that you might choose to use to try to solve the optimization problem in terms of the Alphas.

That's the L1 norm soft margin SVM, and this is the change the algorithm that lets us handle non-linearly separable data sets as well as single outliers that may still be linearly separable but you may choose not to separate [inaudible]. Questions about this? Raise your hand if this stuff makes sense at all. Great.

So the last thing I want to do is talk about an algorithm for actually solving this optimization problem. We wrote down this dual optimization problem with convergence criteria, so let's come up with an efficient algorithm to actually solve this optimization problem. I want to do this partly to give me an excuse to talk about an algorithm called coordinate assent, which is useful to do.

What I actually want to do is tell you about an algorithm called coordinate assent, which is a useful algorithm to know about, and it'll turn out that it won't apply in the simplest form to this problem, but we'll then be able to modify it slightly and then it'll give us a very efficient algorithm for solving this [inaudible] optimization problem. That was the other reason that I had to derive the dual, not only so that we could use kernels but also so that we can apply an algorithm like the SMO algorithm.

First, let's talk about coordinate assent, which is another [inaudible] optimization algorithm. To describe coordinate assent, I just want you to consider the problem of if we want to maximize some function W, which is a function of Alpha one through Alpha M with no constraints. So for now, forget about the constraint that the Alpha [inaudible] must be between zero and C. Forget about the constraint that some of YI Alpha I must be equal to zero. Then this is the coordinate assent algorithm.

It will repeat until convergence and will do for I equals one to M. The [inaudible] of coordinate assent essentially holds all the parameters except Alpha I fixed and then it just maximizes this function with respect to just one of the parameters. Let me write that as Alpha I gets updated as [inaudible] over Alpha I hat of W Alpha one Alpha I minus one. This is really the fancy way of saying hold everything except Alpha I fixed. Just optimize W by optimization objective with respect to only Alpha I. This is just a fancy way of writing it.

This is coordinate assent. One picture that's kind of useful for coordinate assent is if you imagine you're trying to optimize a quadratic function, it really looks like that. These are the contours of the quadratic function and the minimums here. This is what coordinate assent would look like. These are my [inaudible] call this Alpha two and I'll call this Alpha one. My Alpha one Alpha two axis, and so let's say I start down here. Then I'm going to begin by minimizing this with respect to Alpha one. I go there. And then at my new point, I'll minimize with respect to Alpha two, and so I might go to someplace like that.

Then, I'll minimize with respect to Alpha one goes back to Alpha two and so on. You're always taking these axis-aligned steps to get to the minimum. It turns out that there's a modification to this. There are variations of this as well. The way I describe the algorithm, we're always doing this in alternating order. We always optimize with respect to Alpha one then Alpha two, then Alpha one, then Alpha two. What I'm about to say applies only in higher dimensions, but it turns out if you have a lot of parameters, Alpha one through Alpha M, you may not choose to always visit them in a fixed order.

You may choose which Alphas update next depending on what you think will allow you to make the most progress. If you have only two parameters, it makes sense to alternate between them. If you have higher dimensional parameters, sometimes you may choose to update them in a different order if you think doing so would let you make faster progress towards the maximum.

It turns out that coordinate assent compared to some of the algorithms we saw previously – compared

to, say, Newton's method, coordinate assent will usually take a lot more steps, but the chief advantage of coordinate assent when it works well is that sometimes the optimization objective W sometimes is very inexpensive to optimize W with respect to any one of your parameters, and so coordinate assent has to take many more iterations than, say, Newton's method in order to converge.

It turns out that there are many optimization problems for which it's particularly easy to fix all but one of the parameters and optimize with respect to just that one parameter, and if that's true, then the inner group of coordinate assent with optimizing with respect to Alpha I can be done very quickly and cause [inaudible]. It turns out that this will be true when we modify this algorithm to solve the SVM optimization problem. Questions about this? Okay.

Let's go ahead and apply this to our support vector machine dual optimization problem. It turns out that coordinate assent in its basic form does not work for the following reason. The reason is we have constrains on the Alpha Is. Mainly, what we can recall from what we worked out previously, we have a constraint that the sum of [inaudible] Y Alpha I must be equal to zero, and so if you fix all the Alphas except for one, then you can't change one Alpha without violating the constraint.

If I just try to change Alpha one, Alpha one is actually exactly determined as a function of the other Alphas because this was sum to zero. The SMO algorithm, by the way, is due to John Platt, a colleague at Microsoft. The SMO algorithm, therefore, instead of trying to change one Alpha at a time, we will try to change two Alphas at a time. This is called the SMO algorithm, in a sense the sequential minimal optimization and the term minimal refers to the fact that we're choosing the smallest number of Alpha Is to change at a time, which in this case, we need to change at least two at a time.

So then go ahead and outline the algorithm. We will select two Alphas to change, some Alpha I and Alpha J [inaudible] – it just means a rule of thumb. We'll hold all the Alpha Ks fixed except Alpha I and Alpha J and optimize W [inaudible] Alpha with respect to Alpha I and Alpha J subject to all the constraints. It turns out the key step which I'm going to work out is this one, which is how do you optimize W of Alpha with respect to the two parameters that you just chose to update and subject to the constraints? I'll do that in a second.

You would keep running this algorithm until you have satisfied these convergence criteria up to Epsilon. What I want to do now is describe how to do this [inaudible] – how to optimize W of Alpha with respect to Alpha I and Alpha J, and it turns out that it's because you can do this extremely efficiently that the SMO algorithm works well. The [inaudible] for the SMO algorithm can be done extremely efficiently, so it may take a large number of iterations to converge, but each iteration is very cheap.

Let's talk about that. So in order to derive that step where we update in respect to Alpha I and Alpha J, I'm actually going to use Alpha one and Alpha two as my example. I'm gonna update Alpha one and Alpha two. In general, this could be any Alpha I and Alpha J, but just to make my notation on the board easier, I'm going to derive the derivation for Alpha one and Alpha two and the general [inaudible] completely analogous.

On every step of the algorithm with respect to constraint, that sum over I Alpha I YI is equal to zero. This is one of the constraints we had previously for our dual optimization problem. This means that Alpha one Y1 plus Alpha two Y2 must be equal to this, to which I'm going to denote by Zeta. So we also have the constraint that the Alpha Is must be between zero and C. We had two constraints on our dual. This was one of the constraints. This was the other one.

In pictures, the constraint that the Alpha Is is between zero and C, that is often called the Bosk constraint, and so if I draw Alpha one and Alpha two, then I have a constraint that the values of Alpha one and Alpha two must lie within this box that ranges from zero to C. And so the picture of the algorithm is as follows. I have some constraint that Alpha one Y1 plus Alpha two Y2 must equal to Zeta, and so this implies that Alpha one must be equal to Zeta minus Alpha two Y2 over Y1, and so what I want to do is I want to optimize the objective with respect to this.

What I can do is plug in my definition for Alpha one as a function of Alpha two and so this becomes W of Alpha one must be equal to Zeta minus Alpha two Y2 over Y1, Alpha two, Alpha three and so on, and it turns out that because W is a quadratic function – if you look back to our earlier definition of W, you find it's a quadratic function in all the Alphas – it turns out that if you look at this expression for W and if you

view it as just a function of Alpha two, you find that this is a one dimensional quadratic function of Alpha two if you hold Alpha three, Alpha four and so on fixed, and so this can be simplified to some expression of the form A Alpha two squared plus B Alpha two plus C.

This is a standard quadratic function. This is really easy to optimize. We know how to optimize – when did we learn this? This was high school or undergrad or something. You know how to optimize quadratic functions like these. You just do that and that gives you the optimal value for Alpha two. The last step with a Bosk constraint like this – just in pictures, you know your solution must lie on this line, and so there'll be some sort of quadratic function over this line, say, and so if you minimize the quadratic function, maybe you get a value that lies in the box, and if so, then you're done.

Or if your quadratic function looks like this, maybe when you optimize your quadratic function, you may end up with a value outside, so you end up with a solution like that. If that happens, you clip your solution just to map it back inside the box. That'll give you the optimal solution of this quadratic optimization problem subject to your solution satisfying this box constraint and lying on this straight line – in other words, subject to the solution lying on this line segment within the box.

Having solved the Alpha two this way, you can clip it if necessary to get it back within the box constraint and then we have Alpha one as a function of Alpha two and this allows you to optimize W with respect to Alpha one and Alpha two quickly, subject to all the constraints, and the key step is really just sort of one dequadratic optimization, which we do very quickly, which is what makes the inner loop of the SMO algorithm very efficient.

Student: You mentioned here that we can change whatever, but the SMO algorithm, we can change two at a time, so how is that [inaudible] understand that.

Instructor (Andrew Ng):Right. Let's say I want to change – as I run optimization algorithm, I have to respect the constraint that sum over I Alpha I YI must be equal to zero, so this is a linear constraint that I didn't have when I was talking about [inaudible] ascent. Suppose I tried to change just Alpha one. Then I know that Alpha one must be equal to the sum from I equals two to M Alpha I YI divided by Y1, right, and so Alpha one can actually be written exactly as a function of Alpha two, Alpha three and so on through Alpha M. And so if I hold Alpha two, Alpha three, Alpha four through Alpha M fixed, then I can't change Alpha one, because Alpha one is the final [inaudible].

Whereas in contrast, if I choose to change Alpha one and Alpha two at the same time, then I still have a constraint and so I know Alpha one and Alpha two must satisfy that linear constraint but at least this way I can change Alpha one if I also change Alpha two accordingly to make sure this satisfies the constraint.

Student:[Inaudible].

Instructor (Andrew Ng):So Zeta was defined [inaudible]. So on each iteration, I have some setting of the parameters, Alpha one, Alpha two, Alpha three and so on, and I want to change Alpha one and Alpha two, say. So from the previous iteration, let's say I had not validated the constraint, so that holds true, and so I'm just defining Zeta to be equal to this, because Alpha one Y1 plus Alpha two Y2 must be equal to sum from I equals [inaudible] to M of that, and so I'm just defining this to be Zeta.

Student:[Inaudible].

Instructor (Andrew Ng):On every iteration, you change maybe a different pair of Alphas to update. The way you do this is something I don't want to talk about. I'll say a couple more words about that, but the basic outline of the algorithm is on every iteration of the algorithm, you're going to select some Alpha I and Alpha J to update like on this board.

So that's an Alpha I and an Alpha J to update via some [inaudible] and then you use the procedure I just described to actually update Alpha I and Alpha J. What I actually just talked about was the procedure to optimize W with respect to Alpha I and Alpha J. I didn't actually talk about the [inaudible] for choosing Alpha I and Alpha J.

Student: What is the function MW?

Instructor (Andrew Ng):MW is way up there. I'll just write it again. W of Alpha is that function we had previously. W of Alpha was the sum over I – this is about solving the – it was that thing. All of this is about solving the optimization problem for the SVM, so this is the objective function we had, so that's W of Alpha.

Student:[Inaudible]? Exchanging one of the Alphas – optimizing that one, you can make the other one that you have to change work, right?

Instructor (Andrew Ng): What do you mean works?

Student: It will get farther from its optimal.

Instructor (Andrew Ng): Let me translate it differently. What we're trying to do is we're trying to optimize the objective function W of Alpha, so the metric of progress that we care about is whether W of Alpha is getting better on every iteration, and so what is true for coordinate assent and for SMO is on every iteration; W of Alpha can only increase. It may stay the same or it may increase. It can't get worse.

It's true that eventually, the Alphas will converge at some value. It's true that in intervening iterations, one of the Alphas may move further away and then closer and further and closer to its final value, but what we really care about is that W of Alpha is getting better every time, which is true.

Just a couple more words on SMO before I wrap up on this. One is that John Platt's original algorithm talked about a [inaudible] for choosing which values or pairs, Alpha I and Alpha J, to update next is one of those things that's not conceptually complicated but it's very complicated to explain in words.

I won't talk about that here. If you want to learn about it, go ahead and look up John Platt's paper on the SMO algorithm. The [inaudible] is pretty easy to read, and later on, we'll also posting a handout on the course homepage with some of a simplified version of this [inaudible] that you can use in problems. You can see some of the process readings in more details.

One other thing that I didn't talk about was how to update the parameter B. So this is solving all your Alphas. This is also the Alpha that allows us to get W. The other thing I didn't talk about was how to compute the parameter B, and it turns out that again is actually not very difficult. I'll let you read about that yourself with the notes that we'll post along with the next problems.

To wrap up today's lecture, what I want to do is just tell you briefly about a couple of examples of applications of SVMs. Let's consider the problem of Handler's Integer Recognition. In Handler's Integer Recognition, you're given a pixel array with a scanned image of, say, a zip code somewhere in Britain. This is an array of pixels, and some of these pixels will be on and other pixels will be off. This combination of pixels being on maybe represents the character one. The question is given an input feature vector like this, if you have, say, ten pixels by ten pixels, then you have a hundred dimensional feature vector, then [inaudible].

If you have ten pixels by ten pixels, you have 100 features, and maybe these are binary features of XB01 or maybe the Xs are gray scale values corresponding to how dark each of these pixels was. [Inaudible]. Turns out for many years, there was a neuronetwork that was a champion algorithm for Handler's Integer Recognition. And it turns out that you can apply an SVM with the following kernel. It turns out either the polynomial kernel or the Galcean kernel works fine for this problem, and just by writing down this kernel and throwing an SVM at it, an SVM gave performance comparable to the very best neuronetworks.

This is surprising because support vector machine doesn't take into account any knowledge about the pixels, and in particular, it doesn't know that this pixel is next to that pixel because it's just representing the pixel intensity value as a vector. And so this means the performance of SVM would be the same even if you were to shuffle all the pixels around. [Inaudible] let's say comparable to the very best neuronetworks, which had been under very careful development for many years.

I want to tell you about one other cool example, which is SVMs are also used also to classify other fairly esoteric objects. So for example, let's say you want to classify protein sequences into different classes of proteins. Every time I do this, I suspect that biologists in the room cringe, so I apologize for that. There are 20 amino acids, and proteins in our bodies are made up by sequences of amino acids. Even though there are 20 amino acids and 26 alphabets, I'm going to denote amino acids by the alphabet A through Z with apologizes to the biologists.

Here's an amino acid sequence represented by a series of alphabets. So suppose I want to assign this protein into a few classes depending on what type of protein it is. The question is how do I construct my feature vector? This is challenging for many reasons, one of which is that protein sequences can be of different lengths. There are some very long protein sequences and some very short ones, and so you can't have a feature saying what is the amino acid in the 100th position, because maybe there is no 100th position in this protein sequence. Some are very long. Some are very short.

Here's my feature representation, which is I'm going to write down all possible combinations of four alphabets. I'm going to write down AAAA, AAAB, AAAC down to AAAZ and then AABA and so on. You get the idea. Write down all possible combinations of four alphabets and my feature representation will be I'm going to scan through this sequence of amino acids and count how often each of these subsequences occur. So for example, BAJT occurs twice and so I'll put a two there, and none of these sequences occur, so I'll put a zero there. I guess I have a one here and a zero there.

This very long vector will be my feature representation for protein. This representation applies no matter how long my protein sequence is. How large is this? Well, it turns out this is going to be in R20 to the four, and so you have a 160,000 dimensional feature vector, which is reasonably large, even by modern computer standards. Clearly, we don't want to explicitly represent these high dimensional feature vectors. Imagine you have 1,000 examples and you store this as double [inaudible]. Even on modern day computers, this is big.

It turns out that there's an efficient dynamic programming algorithm that can efficiently compute inner products between these feature vectors, and so we can apply this feature representation, even though it's a ridiculously high feature vector to classify protein sequences. I won't talk about the [inaudible] algorithm. If any of you have seen the [inaudible] algorithm for finding subsequences, it's kind of reminiscent of that. You can look those up if you're interested.

This is just another example of a cool kernel, and more generally, if you're faced with some new machine-learning problem, sometimes you can choose a standard kernel like a Galcean kernel, and sometimes there are research papers written on how to come up with a new kernel for a new problem. Two last sentences I want to say. Where are we now? That wraps up SVMs, which many people would consider one of the most effective off the shelf learning algorithms, and so as of today, you've actually seen a lot of learning algorithms.

I want to close this class by saying congrats. You're now well qualified to actually go and apply learning algorithms to a lot of problems. We're still in week four of the quarter, so there's more to come. In particular, what I want to do next is talk about how to really understand the learning algorithms and when they work well and when they work poorly and to take the tools which you now have and really talk about how you can use them really well. We'll start to do that in the next lecture. Thanks.

[End of Audio]
Duration: 78 minutes

1.9 Machine Learning Lecture 9[9]

Figure 1.9

MachineLearning-Lecture09

Instructor (Andrew Ng): All right, good morning. Just one quick announcement, first things for all of your project proposals, I've read through all of them and they all look fine. There is one or two that I was trying to email back comments on that had slightly questionable aspects, but if you don't hear by me from today you can safely assume that your project proposal is fine and you should just go ahead and start working on your proposals. You should just go ahead and start working on your project. Okay, there's many exciting proposals sent in on Friday and so I think the proposal session at the end of the quarter will be an exciting event.

---
[9]This content is available online at <http://cnx.org/content/m45916/1.2/>.

Okay. So welcome back. What I want to do today is start a new chapter in between now and then. In particular, I want to talk about learning theory. So in the previous, I guess eight lectures so far, you've learned about a lot of learning algorithms, and yes, you now I hope understand a little about some of the best and most powerful tools of machine learning in the [inaudible]. And all of you are now sort of well qualified to go into industry and though powerful learning algorithms apply, really the most powerful learning algorithms we know to all sorts of problems, and in fact, I hope you start to do that on your projects right away as well.

You might remember, I think it was in the very first lecture, that I made an analogy to if you're trying to learn to be a carpenter, so if you imagine you're going to carpentry school to learn to be a carpenter, then only a small part of what you need to do is to acquire a set of tools. If you learn to be a carpenter you don't walk in and pick up a tool box and [inaudible], so when you need to cut a piece of wood do you use a rip saw, or a jig saw, or a keyhole saw whatever, is this really mastering the tools there's also an essential part of becoming a good carpenter. And what I want to do in the next few lectures is actually give you a sense of the mastery of the machine learning tools all of you have. Okay?

And so in particular, in the next few lectures what I want to is to talk more deeply about the properties of different machine learning algorithms so that you can get a sense of when it's most appropriate to use each one. And it turns out that one of the most common scenarios in machine learning is someday you'll be doing research or [inaudible] a company. And you'll apply one of the learning algorithms you learned about, you may apply logistic regression, or support vector machines, or Naïve Bayes or something, and for whatever bizarre reason, it won't work as well as you were hoping, or it won't quite do what you were hoping it to.

To me what really separates the people from – what really separates the people that really understand and really get machine learning, compared to people that maybe read the textbook and so they'll work through the math, will be what you do next. Will be in your decisions of when you apply a support vector machine and it doesn't quite do what you wanted, do you really understand enough about support vector machines to know what to do next and how to modify the algorithm? And to me that's often what really separates the great people in machine learning versus the people that like read the text book and so they'll [inaudible] the math, and so they'll have just understood that. Okay? So what I want to do today – today's lecture will mainly be on learning theory and we'll start to talk about some of the theoretical results of machine learning. The next lecture, later this week, will be on algorithms for sort of [inaudible], or fixing some of the problems that the learning theory will point out to us and help us understand. And then two lectures from now, that lecture will be almost entirely focused on the practical advice for how to apply learning algorithms. Okay? So you have any questions about this before I start? Okay.

So the very first thing we're gonna talk about is something that you've probably already seen on the first homework, and something that alluded to previously, which is the bias variance trade-off. So take ordinary least squares, the first learning algorithm we learned about, if you [inaudible] a straight line through these datas, this is not a very good model. Right. And if this happens, we say it has underfit the data, or we say that this is a learning algorithm with a very high bias, because it is failing to fit the evident quadratic structure in the data. And for the prefaces of [inaudible] you can formally think of the bias of the learning algorithm as representing the fact that even if you had an infinite amount of training data, even if you had tons of training data, this algorithm would still fail to fit the quadratic function – the quadratic structure in the data. And so we think of this as a learning algorithm with high bias. Then there's the opposite problem, so that's the same dataset. If you fit a fourth of the polynomials into this dataset, then you have – you'll be able to interpolate the five data points exactly, but clearly, this is also not a great model to the structure that you and I probably see in the data.

And we say that this algorithm has a problem – excuse me, is overfitting the data, or alternatively that this algorithm has high variance. Okay? And the intuition behind overfitting a high variance is that the algorithm is fitting serious patterns in the data, or is fitting idiosyncratic properties of this specific dataset, be it the dataset of housing prices or whatever. And quite often, they'll be some happy medium of fitting a quadratic function that maybe won't interpolate your data points perfectly, but also captures multi-structure in your data than a simple model which under fits. I say that you can sort of have the exactly the same picture of classification problems as well, so lets say this is my training set, right, of positive and negative

examples, and so you can fit logistic regression with a very high order polynomial [inaudible], or [inaudible] of X equals the sigmoid function of – whatever. Sigmoid function applied to a tenth of the polynomial. And you do that, maybe you get a decision boundary like this. Right. That does indeed perfectly separate the positive and negative classes, this is another example of how overfitting, and in contrast you fit logistic regression into this model with just the linear features, with none of the quadratic features, then maybe you get a decision boundary like that, which can also underfit. Okay.

So what I want to do now is understand this problem of overfitting versus underfitting, of high bias versus high variance, more explicitly, I will do that by posing a more formal model of machine learning and so trying to prove when these two twin problems – when each of these two problems come up. And as I'm modeling the example for our initial foray into learning theory, I want to talk about learning classification, in which H of X is equal to G of data transpose X. Okay? So the learning classifier. And for this class I'm going to use, Z – excuse me – I'm gonna use G as indicator Z grading with zero. With apologies in advance for changing the notation yet again, for the support vector machine lectures we use Y equals minus one or plus one. For learning theory lectures, turns out it'll be a bit cleaner if I switch back to Y equals zero-one again, so I'm gonna switch back to my original notation. And so you think of this model as a model forum as logistic regressions, say, and think of this as being similar to logistic regression, except that now we're going to force the logistic regression algorithm, to opt for labels that are either zero or one. Okay? So you can think of this as a classifier to opt for labels zero or one involved in the probabilities. And so as usual, let's say we're given a training set of M examples. That's just my notation for writing a set of M examples ranging from I equals one through M. And I'm going to assume that the training example is XIYI. I've drawn IID, from sum distribution, scripts D. Okay? [Inaudible]. Identically and definitively distributed and if you have – you have running a classification problem on houses, like features of the house comma, whether the house will be sold in the next six months, then this is just the priority distribution over features of houses and whether or not they'll be sold. Okay?

So I'm gonna assume that training examples we've drawn IID from some probability distributions, scripts D. Well, same thing for spam, if you're trying to build a spam classifier then this would be the distribution of what emails look like comma, whether they are spam or not. And in particular, to understand or simplify – to understand the phenomena of bias invariance, I'm actually going to use a simplified model of machine learning. And in particular, logistic regression fits this parameters the model like this for maximizing the law of likelihood. But in order to understand learning algorithms more deeply, I'm just going to assume a simplified model of machine learning, let me just write that down. So I'm going to define training error as – so this is a training error of a hypothesis X subscript data. Write this epsilon hat of subscript data. If I want to make the dependence on a training set explicit, I'll write this with a subscript S there where S is a training set. And I'll define this as, let's see. Okay. I hope the notation is clear. This is a sum of indicator functions for whether your hypothesis correctly classifies the Y – the IFE example.

And so when you divide by M, this is just in your training set what's the fraction of training examples your hypothesis classifies so defined as a training error. And training error is also called risk. The simplified model of machine learning I'm gonna talk about is called empirical risk minimization. And in particular, I'm going to assume that the way my learning algorithm works is it will choose parameters data, that minimize my training error. Okay? And it will be this learning algorithm that we'll prove properties about. And it turns out that you can think of this as the most basic learning algorithm, the algorithm that minimizes your training error.

It turns out that logistic regression and support vector machines can be formally viewed as approximation cities, so it turns out that if you actually want to do this, this is a nonconvex optimization problem. This is actually – it actually [inaudible] hard to solve this optimization problem. And logistic regression and support vector machines can both be viewed as approximations to this nonconvex optimization problem by finding the convex approximation to it. Think of this as similar to what algorithms like logistic regression are doing. So let me take that definition of empirical risk minimization and actually just rewrite it in a different equivalent way. For the results I want to prove today, it turns out that it will be useful to think of our learning algorithm as not choosing a set of parameters, but as choosing a function. So let me say what I mean by that. Let me define the hypothesis class, script h, as the class of all hypotheses of – in other words

as the class of all linear classifiers, that your learning algorithm is choosing from. Okay? So H subscript data is a specific linear classifier, so H subscript data, in each of these functions – each of these is a function mapping from the input domain X is the class zero-one. Each of these is a function, and as you vary the parameter's data, you get different functions. And so let me define the hypothesis class script H to be the class of all functions that say logistic regression can choose from. Okay.

So this is the class of all linear classifiers and so I'm going to define, or maybe redefine empirical risk minimization as instead of writing this choosing a set of parameters, I want to think of it as choosing a function into hypothesis class of script H that minimizes – that minimizes my training error. Okay? So – actually can you raise your hand if it makes sense to you why this is equivalent to the previous formulation? Okay, cool. Thanks. So for development of the use of think of algorithms as choosing from function from the class instead, because in a more general case this set, script H, can be some other class of functions. Maybe is a class of all functions represented by viewer network, or the class of all – some other class of functions the learning algorithm wants to choose from. And this definition for empirical risk minimization will still apply. Okay? So what we'd like to do is understand whether empirical risk minimization is a reasonable algorithm. Alex?

Student:[Inaudible] a function that's defined by G of data TX, or is it now more general?

Instructor (Andrew Ng): I see, right, so lets see – I guess this – the question is H data still defined by G of phase transpose X, is this more general? So –

Student:[Inaudible]

Instructor (Andrew Ng): Oh, yeah so very – two answers to that. One is, this framework is general, so for the purpose of this lecture it may be useful to you to keep in mind a model of the example of when H subscript data is the class of all linear classifiers such as those used by like a visectron algorithm or logistic regression. This – everything on this board, however, is actually more general. H can be any set of functions, mapping from the INFA domain to the center of class label zero and one, and then you can perform empirical risk minimization over any hypothesis class.

For the purpose of today's lecture, I am going to restrict myself to talking about binary classification, but it turns out everything I say generalizes to regression in other problem as well. Does that answer your question?

Student: Yes.

Instructor (Andrew Ng): Cool. All right. So I wanna understand if empirical risk minimization is a reasonable algorithm. In particular, what are the things we can prove about it? So clearly we don't actually care about training error, we don't really care about making accurate predictions on the training set, or at a least that's not the ultimate goal. The ultimate goal is how well it makes – generalization – how well it makes predictions on examples that we haven't seen before. How well it predicts prices or sale or no sale outcomes of houses you haven't seen before.

So what we really care about is generalization error, which I write as epsilon of H. And this is defined as the probability that if I sample a new example, X comma Y, from that distribution scripts D, my hypothesis mislabels that example. And in terms of notational convention, usually if I use – if I place a hat on top of something, it usually means – not always – but it usually means that it is an attempt to estimate something about the hat. So for example, epsilon hat here – this is something that we're trying – think of epsilon hat training error as an attempt to approximate generalization error. Okay, so the notation convention is usually the things with the hats on top are things we're using to estimate other quantities. And H hat is a hypothesis output by learning algorithm to try to estimate what the functions from H to Y – X to Y. So let's actually prove some things about when empirical risk minimization will do well in a sense of giving us low generalization error, which is what we really care about. In order to prove our first learning theory result, I'm going to have to state two lemmas, the first is the union vowel, which is the following, let A1 through AK be K event. And when I say events, I mean events in a sense of a probabilistic event that either happens or not. And these are not necessarily independent.

So there's some current distribution over the events A one through AK, and maybe they're independent maybe not, no assumption on that. Then the probability of A one or A two or dot, dot, dot, up to AK, this union symbol, this hat, this just means this sort of just set notation for probability just means "or." So

the probability of at least one of these events occurring, of A one or A two, or up to AK, this is S equal to the probability of A one plus probability of A two plus dot, dot, dot, plus probability of AK. Okay? So the intuition behind this is just that – I'm not sure if you've seen Venn diagrams depictions of probability before, if you haven't, what I'm about to do may be a little cryptic, so just ignore that. Just ignore what I'm about to do if you haven't seen it before. But if you have seen it before then this is really – this is really great – the probability of A one, union A two, union A three, is less than the P of A one, plus P of A two, plus P of A three. Right. So that the total mass in the union of these three things [inaudible] to the sum of the masses in the three individual sets, it's not very surprising.

It turns out that depending on how you define your axioms of probability, this is actually one of the axioms that probably varies, so I won't actually try to prove this. This is usually written as an axiom. So sigmas of avitivity are probably measured as this – what is sometimes called as well. But in learning theory it's commonly called the union balance – I just call it that. The other lemma I need is called the Hufting inequality. And again, I won't actually prove this, I'll just state it, which is – let's let Z1 up to ZM, BM, IID, there may be random variables with mean Phi. So the probability of ZI equals 1 is equal to Phi. So let's say you observe M IID for newly random variables and you want to estimate their mean. So let me define Phi hat, and this is again that notation, no convention, Phi hat means – does not attempt – is an estimate or something else. So when we define Phi hat to be 1 over M, semper my equals one through MZI. Okay? So this is our attempt to estimate the mean of these Benuve random variables by sort of taking its average. And let any gamma be fixed.

Then, the Hufting inequality is that the probability your estimate of Phi is more than gamma away from the true value of Phi, that this is bounded by two E to the next of two gamma squared. Okay? So just in pictures – so this theorem holds – this lemma, the Hufting inequality, this is just a statement of fact, this just holds true. But let me now draw a cartoon to describe some of the intuition behind this, I guess. So lets say [inaudible] this is a real number line from zero to one. And so Phi is the mean of your Benuve random variables. You will remember from – you know, whatever – some undergraduate probability or statistics class, the central limit theorem that says that when you average all the things together, you tend to get a Gaussian distribution. And so when you toss M coins with bias Phi, we observe these M Benuve random variables, and we average them, then the probability distribution of Phi hat will roughly be a Gaussian lets say. Okay? It turns out if you haven't seen this up before, this is actually that the cumulative distribution function of Phi hat will converse with that of the Gaussian. Technically Phi hat can only take on a discreet set of values because these are factions one over Ms. It doesn't really have an entity but just as a cartoon think of it as a converse roughly to a Gaussian.

So what the Hufting inequality says is that if you pick a value of gamma, let me put S one interval gamma there's another interval gamma. Then the saying that the probability mass of the details, in other words the probability that my value of Phi hat is more than a gamma away from the true value, that the total mass – that the total probability mass in these tails is at most two E to the negative two gamma squared M. Okay? That's what the Hufting inequality – so if you can't read that this just says – this is just the right hand side of the bound, two E to negative two gamma squared. So balance the probability that you make a mistake in estimating the mean of a Benuve random variable.

And the cool thing about this bound – the interesting thing behind this bound is that the [inaudible] exponentially in M, so it says that for a fixed value of gamma, as you increase the size of your training set, as you toss a coin more and more, then the worth of this Gaussian will shrink. The worth of this Gaussian will actually shrink like one over root to M. And that will cause the probability mass left in the tails to decrease exponentially, quickly, as a function of that. And this will be important later. Yeah?

Student:
Does this come from the central limit theorem [inaudible].

Instructor (Andrew Ng): No it doesn't. So this is proved by a different – this is proved – no – so the central limit theorem – there may be a version of the central limit theorem, but the versions I'm familiar with tend – are sort of asymptotic, but this works for any finer value of M. Oh, and for your – this bound holds even if M is equal to two, or M is [inaudible], if M is very small, the central limit theorem approximation is not gonna hold, but this theorem holds regardless. Okay? I'm drawing this just as a cartoon to help explain

the intuition, but this theorem just holds true, without reference to central limit theorem.

All right. So lets start to understand empirical risk minimization, and what I want to do is begin with studying empirical risk minimization for a [inaudible] case that's a logistic regression, and in particular I want to start with studying the case of finite hypothesis classes. So let's say script H is a class of K hypotheses. Right. So this is K functions with no – each of these is just a function mapping from inputs to outputs, there's no parameters in this. And so what the empirical risk minimization would do is it would take the training set and it'll then look at each of these K functions, and it'll pick whichever of these functions has the lowest training error. Okay?

So now that the logistic regression uses an infinitely large – a continuous infinitely large class of hypotheses, script H, but to prove the first row I actually want to just describe our first learning theorem is all for the case of when you have a finite hypothesis class, and then we'll later generalize that into the hypothesis classes. So empirical risk minimization takes the hypothesis of the lowest training error, and what I'd like to do is prove a bound on the generalization error of H hat. All right. So in other words I'm gonna prove that somehow minimizing training error allows me to do well on generalization error.

And here's the strategy, I'm going to – the first step in this prove I'm going to show that training error is a good approximation to generalization error, and then I'm going to show that this implies a bound on the generalization error of the hypothesis of [inaudible] empirical risk minimization. And I just realized, this class I guess is also maybe slightly notation heavy class round, instead of just introducing a reasonably large set of new symbols, so if again, in the course of today's lecture, you're looking at some symbol and you don't quite remember what it is, please raise your hand and ask. [Inaudible] what's that, what was that, was that a generalization error or was it something else? So raise your hand and ask if you don't understand what the notation I was defining.

Okay. So let me introduce this in two steps. And the empirical risk strategy is I'm gonna show training errors that give approximation generalization error, and this will imply that minimizing training error will also do pretty well in terms of minimizing generalization error. And this will give us a bound on the generalization error of the hypothesis output by empirical risk minimization. Okay? So here's the idea. So lets even not consider all the hypotheses at once, lets pick any hypothesis, HJ in the class script H, and so until further notice lets just consider there one fixed hypothesis. So pick any one hypothesis and let's talk about that one.

Let me define ZI to be indicator function for whether this hypothesis misclassifies the IFE example – excuse me – or Z subscript I. Okay? So ZI would be zero or one depending on whether this one hypothesis which is the only one I'm gonna even consider now, whether this hypothesis was classified as an example. And so my training set is drawn randomly from sum distribution scripts d, and depending on what training examples I've got, these ZIs would be either zero or one. So let's figure out what the probability distribution ZI is. Well, so ZI takes on the value of either zero or one, so clearly is a Benuve random variable, it can only take on these values.

Well, what's the probability that ZI is equal to one? In other words, what's the probability that from a fixed hypothesis HJ, when I sample my training set IID from distribution D, what is the chance that my hypothesis will misclassify it? Well, by definition, that's just a generalization error of my hypothesis HJ. So ZI is a Benuve random variable with mean given by the generalization error of this hypothesis. Raise your hand if that made sense. Oh, cool. Great.

And moreover, all the ZIs have the same probability of being one, and all my training examples I've drawn are IID, and so the ZIs are also independent – and therefore the ZIs themselves are IID random variables. Okay? Because my training examples were drawn independently of each other, by assumption.

If you read this as the definition of training error, the training error of my hypothesis HJ, that's just that. That's just the average of my ZIs, which was – well I previously defined it like this. Okay? And so epsilon hat of HJ is exactly the average of MIID, Benuve random variables, drawn from Benuve distribution with mean given by the generalization error, so this is – well this is the average of MIID Benuve random variables, each of which has meaning given by the generalization error of HJ.

And therefore, by the Hufting inequality we have to add the probability that the difference between training and generalization error, the probability that this is greater than gamma is less than to two, E to

the negative two, gamma squared M. Okay? Exactly by the Hufting inequality.

And what this proves is that, for my fixed hypothesis HJ, my training error, epsilon hat will with high probability, assuming M is large, if M is large than this thing on the right hand side will be small, because this is two Es and a negative two gamma squared M. So this says that if my training set is large enough, then the probability my training error is far from generalization error, meaning that it is more than gamma, will be small, will be bounded by this thing on the right hand side. Okay?

Now, here's the [inaudible] tricky part, what we've done is approve this bound for one fixed hypothesis, for HJ. What I want to prove is that training error will be a good estimate for generalization error, not just for this one hypothesis HJ, but actually for all K hypotheses in my hypothesis class script H. So let's do it – well, better do it on a new board. So in order to show that, let me define a random event, let me define AJ to be the event that – let me define AJ to be the event that – you know, the difference between training and generalization error is more than gamma on a hypothesis HJ. And so what we put on the previous board was that the probability of AJ is less equal to two E to the negative two, gamma squared M, and this is pretty small. Now, What I want to bound is the probability that there exists some hypothesis in my class script H, such that I make a large error in my estimate of generalization error. Okay? Such that this holds true. So this is really just that the probability that there exists a hypothesis for which this holds. This is really the probability that A one or A two, or up to AK holds. The chance there exists a hypothesis is just well the priority that – for hypothesis one and make a large error in estimating the generalization error, or for hypothesis two and make a large error in estimating generalization error, and so on. And so by the union bound, this is less than equal to that, which is therefore less than equal to – is equal to that. Okay?

So let me just take one minus both sides – of the equation on the previous board – let me take one minus both sides, so the probability that there does not exist for hypothesis such that, that. The probability that there does not exist a hypothesis on which I make a large error in this estimate while this is equal to the probability that for all hypotheses, I make a small error, or at most gamma, in my estimate of generalization error. In taking one minus on the right hand side I get two KE to the negative two gamma squared M. Okay? And so – and the sign of the inequality flipped because I took one minus both sides. The minus sign flips the sign of the equality. So what we're shown is that with probability – which abbreviates to WP – with probability one minus two KE to the negative two gamma squared M. We have that, epsilon hat of H will be – will then gamma of epsilon of H, simultaneously for all hypotheses in our class script H.

And so just to give this result a name, this is called – this is one instance of what's called a uniform conversions result, and the term uniform conversions – this sort of alludes to the fact that this shows that as M becomes large, then these epsilon hats will all simultaneously converge to epsilon of H. That training error will become very close to generalization error simultaneously for all hypotheses H. That's what the term uniform refers to, is the fact that this converges for all hypotheses H and not just for one hypothesis. And so what we're shown is one example of a uniform conversions result. Okay? So let me clean a couple more boards. I'll come back and ask what questions you have about this. We should take another look at this and make sure it all makes sense. Yeah, okay. What questions do you have about this?

Student:
How the is the value of gamma computed [inaudible]?

Instructor (Andrew Ng): Right. Yeah. So let's see, the question is how is the value of gamma computed? So for these purposes – for the purposes of this, gamma is a constant. Imagine a gamma is some constant that we chose in advance, and this is a bound that holds true for any fixed value of gamma. Later on as we take this bound and then sort of develop this result further, we'll choose specific values of gamma as a [inaudible] of this bound. For now we'll just imagine that when we're proved this holds true for any value of gamma. Any questions? Yeah?

Student:[Inaudible] hypothesis phase is infinite [inaudible]?

Instructor (Andrew Ng): Yes, the labs in the hypothesis phase is infinite, so this simple result won't work in this present form, but we'll generalize this – probably won't get to it today – but we'll generalize this at the beginning of the next lecture to infinite hypothesis classes.

Student: How do we use this theory [inaudible]?

Instructor (Andrew Ng): How do you use theorem factors? So let me – I might get to a little of that

later today, we'll talk concretely about algorithms, the consequences of the understanding of these things in the next lecture as well. Yeah, okay? Cool. Can you just raise your hand if the things I've proved so far make sense? Okay. Cool. Great. Thanks.

All right. Let me just take this uniform conversions bound and rewrite it in a couple of other forms. So this is a sort of a bound on probability, this is saying suppose I fix my training set and then fix my training set – fix my threshold, my error threshold gamma, what is the probability that uniform conversions holds, and well, that's my formula that gives the answer. This is the probability of something happening.

So there are actually three parameters of interest. One is, "What is this probability?" The other parameter is, "What's the training set size M?" And the third parameter is, "What is the value of this error threshold gamma?" I'm not gonna vary K for these purposes. So other two other equivalent forms of the bounds, which – so you can ask, "Given gamma – so what we proved was given gamma and given M, what is the probability of uniform conversions?" The other equivalent forms are, so that given gamma and the probability delta of making a large error, how large a training set size do you need in order to give a bound on – how large a training set size do you need to give a uniform conversions bound with parameters gamma and delta?

And so – well, so if you set delta to be two KE so negative two gamma squared M. This is that form that I had on the left. And if you solve for M, what you find is that there's an equivalent form of this result that says that so long as your training set assigns M as greater than this. And this is the formula that I get by solving for M. Okay? So long as M is greater than equal to this, then with probability, which I'm abbreviating to WP again, with probability at least one minus delta, we have for all. Okay? So this says how large a training set size that I need to guarantee that with probability at least one minus delta, we have the training error is within gamma of generalization error for all my hypotheses, and this gives an answer.

And just to give this another name, this is an example of a sample complexity bound. So from undergrad computer science classes you may have heard of computational complexity, which is how much computations you need to achieve something. So sample complexity just means how large a training example – how large a training set – how large a sample of examples do you need in order to achieve a certain bound and error. And it turns out that in many of the theorems we write out you can pose them in sort of a form of probability bound or a sample complexity bound or in some other form. I personally often find the sample complexity bounds the most easy to interpret because it says how large a training set do you need to give a certain bound on the errors.

And in fact – well, we'll see this later, sample complexity bounds often sort of help to give guidance for really if you're trying to achieve something on a machine learning problem, this really is trying to give guidance on how much training data you need to prove something.

The one thing I want to note here is that M grows like the log of K, right, so the log of K grows extremely slowly as a function of K. The log is one of the slowest growing functions, right. It's one of – well, some of you may have heard this, right? That for all values of K, right – I learned this from a colleague, Andrew Moore, at Carnegie Mellon – that in computer science for all practical purposes for all values of K, log K is less [inaudible], this is almost true. So log K is – logs is one of the slowest growing functions, and so the fact that M sample complexity grows like the log of K, means that you can increase this number of hypotheses in your hypothesis class quite a lot and the number of the training examples you need won't grow very much.

[Inaudible]. This property will be important later when we talk about infinite hypothesis classes. The final form is the – I guess is sometimes called the error bound, which is when you hold M and delta fixed and solved for gamma. And so – and what do you do – what you get then is that the probability at least one minus delta, we have that. For all hypotheses in my hypothesis class, the difference in the training generalization error would be less than equal to that. Okay? And that's just solving for gamma and plugging the value I get in there. Okay?

All right. So the second step of the overall proof I want to execute is the following. The result of the training error is essentially that uniform conversions will hold true with high probability. What I want to show now is let's assume that uniform conversions hold. So let's assume that for all hypotheses H, we have that epsilon of H minus epsilon hat of H, is less than of the gamma. Okay? What I want to do now is use this to see what we can prove about the bound of – see what we can prove about the generalization error. So I want to know – suppose this holds true – I want to know can we prove something about the generalization

error of H hat, where again, H hat was the hypothesis selected by empirical risk minimization. Okay?

So in order to show this, let me make one more definition, let me define H star, to be the hypothesis in my class script H that has the smallest generalization error. So this is – if I had an infinite amount of training data or if I really I could go in and find the best possible hypothesis – best possible hypothesis in the sense of minimizing generalization error – what's the hypothesis I would get? Okay? So in some sense, it sort of makes sense to compare the performance of our learning algorithm to the performance of H star, because we sort of – we clearly can't hope to do better than H star. Another way of saying that is that if your hypothesis class is a class of all linear decision boundaries, that the data just can't be separated by any linear functions. So if even H star is really bad, then there's sort of – it's unlikely – then there's just not much hope that your learning algorithm could do even better than H star. So I actually prove this result in three steps. So the generalization error of H hat, the hypothesis I chose, this is going to be less than equal to that, actually let me number these equations, right. This is – because of equation one, because I see that epsilon of H hat and epsilon hat of H hat will then gamma of each other. Now because H star, excuse me, now by the definition of empirical risk minimization, H hat was chosen to minimize training error and so there can't be any hypothesis with lower training error than H hat. So the training error of H hat must be less than the equal to the training error of H star. So this is sort of by two, or by the definition of H hat, as the hypothesis that minimizes training error H hat.

And the final step is I'm going to apply this uniform conversions result again. We know that epsilon hat of H star must be moving gamma of epsilon of H star. And so this is at most plus gamma. Then I have my original gamma there. Okay? And so this is by equation one again because – oh, excuse me – because I know the training error of H star must be moving gamma of the generalization error of H star. And so – well, I'll just write this as plus two gamma. Okay? Yeah?

Student:[Inaudible] notation, is epsilon proof of [inaudible] H hat that's not the training error, that's the generalization error with estimate of the hypothesis?

Instructor (Andrew Ng): Oh, okay. Let me just – well, let me write that down on this board. So actually – actually let me think – [inaudible] fit this in here. So epsilon hat of H is the training error of the hypothesis H. In other words, given the hypothesis – a hypothesis is just a function, right mapped from X or Ys – so epsilon hat of H is given the hypothesis H, what's the fraction of training examples it misclassifies? And generalization error of H, is given the hypothesis H if I sample another example from my distribution scripts D, what's the probability that H will misclassify that example? Does that make sense?

Student:[Inaudible]?

Instructor (Andrew Ng): Oh, okay. And H hat is the hypothesis that's chosen by empirical risk minimization. So when I talk about empirical risk minimization, is the algorithm that minimizes training error, and so epsilon hat of H is the training error of H, and so H hat is defined as the hypothesis that out of all hypotheses in my class script H, the one that minimizes training error epsilon hat of H. Okay?

All right. Yeah?

Student:[Inaudible] H is [inaudible] a member of typical H, [inaudible] family right?

Instructor (Andrew Ng): Yes it is.

Student: So what happens with the generalization error [inaudible]?

Instructor (Andrew Ng): I'll talk about that later. So let me tie all these things together into a theorem. Let there be a hypothesis class given with a finite set of K hypotheses, and let any M delta be fixed. Then – so I fixed M and delta, so this will be the error bound form of the theorem, right? Then, with probability at least one minus delta. We have that. The generalization error of H hat is less than or equal to the minimum over all hypotheses in set H epsilon of H, plus two times, plus that. Okay?

So to prove this, well, this term of course is just epsilon of H star. And so to prove this we set gamma to equal to that – this is two times the square root term. To prove this theorem we set gamma to equal to that square root term. Say that again?

Student:[Inaudible].

Instructor (Andrew Ng): Wait. Say that again?

Student:[Inaudible].

Instructor (Andrew Ng): Oh, yes. Thank you. That didn't make sense at all. Thanks. Great. So set

gamma to that square root term, and so we know equation one, right, from the previous board holds with probability one minus delta. Right. Equation one was the uniform conversions result right, that – well, IE. This is equation one from the previous board, right, so set gamma equal to this we know that we'll probably use one minus delta this uniform conversions holds, and whenever that holds, that implies – you know, I guess – if we call this equation "star" I guess. And whenever uniform conversions holds, we showed again, on the previous boards that this result holds, that generalization error of H hat is less than two – generalization error of H star plus two times gamma. Okay? And so that proves this theorem.

So this result sort of helps us to quantify a little bit that bias variance tradeoff that I talked about at the beginning of – actually near the very start of this lecture. And in particular let's say I have some hypothesis class script H, that I'm using, maybe as a class of all linear functions and linear regression, and logistic regression with just the linear features. And let's say I'm considering switching to some new class H prime by having more features. So lets say this is linear and this is quadratic, so the class of all linear functions and the subset of the class of all quadratic functions, and so H is the subset of H prime. And let's say I'm considering – instead of using my linear hypothesis class – let's say I'm considering switching to a quadratic hypothesis class, or switching to a larger hypothesis class. Then what are the tradeoffs involved? Well, I proved this only for finite hypothesis classes, but we'll see that something very similar holds for infinite hypothesis classes too. But the tradeoff is what if I switch from H to H prime, or I switch from linear to quadratic functions. Then epsilon of H star will become better because the best hypothesis in my hypothesis class will become better.

The best quadratic function – by best I mean in the sense of generalization error – the hypothesis function – the quadratic function with the lowest generalization error has to have equal or more likely lower generalization error than the best linear function. So by switching to a more complex hypothesis class you can get this first term as you go down. But what I pay for then is that K will increase. By switching to a larger hypothesis class, the first term will go down, but the second term will increase because I now have a larger class of hypotheses and so the second term K will increase.

And so this is sometimes called the bias – this is usually called the bias variance tradeoff. Whereby going to larger hypothesis class maybe I have the hope for finding a better function, that my risk of sort of not fitting my model so accurately also increases, and that's because – illustrated by the second term going up when the size of your hypothesis, when K goes up. And so speaking very loosely, we can think of this first term as corresponding maybe to the bias of the learning algorithm, or the bias of the hypothesis class. And you can – again speaking very loosely, think of the second term as corresponding to the variance in your hypothesis, in other words how well you can actually fit a hypothesis in the – how well you actually fit this hypothesis class to the data. And by switching to a more complex hypothesis class, your variance increases and your bias decreases.

As a note of warning, it turns out that if you take like a statistics class you've seen definitions of bias and variance, which are often defined in terms of squared error or something. It turns out that for classification problems, there actually is no universally accepted formal definition of bias and variance for classification problems. For regression problems, there is this square error definition. For classification problems it turns out there've been several competing proposals for definitions of bias and variance. So when I say bias and variance here, think of these as very loose, informal, intuitive definitions, and not formal definitions. Okay. The cartoon associated with intuition I just said would be as follows: Let's say – and everything about the plot will be for a fixed value of M, for a fixed training set size M. Vertical axis I'll plot ever and on the horizontal axis I'll plot model complexity. And by model complexity I mean sort of degree of polynomial, size of your hypothesis class script H etc. It actually turns out, you remember the bandwidth parameter from locally weighted linear regression, that also has a similar effect in controlling how complex your model is. Model complexity [inaudible] polynomial I guess. So the more complex your model, the better your training error, and so your training error will tend to [inaudible] zero as you increase the complexity of your model because the more complete your model the better you can fit your training set.

But because of this bias variance tradeoff, you find that generalization error will come down for a while and then it will go back up. And this regime on the left is when you're underfitting the data or when you have high bias. And this regime on the right is when you have high variance or you're overfitting the data.

Okay? And this is why a model of sort of intermediate complexity, somewhere here if often preferable to if [inaudible] and minimize generalization error. Okay? So that's just a cartoon. In the next lecture we'll actually talk about the number of algorithms for trying to automatically select model complexities, say to get you as close as possible to this minimum – to this area of minimized generalization error. The last thing I want to do is actually going back to the theorem I wrote out, I just want to take that theorem – well, so the theorem I wrote out was an error bound theorem this says for fixed M and delta where probability one minus delta, I get a bound on gamma, which is what this term is. So the very last thing I wanna do today is just come back to this theorem and write out a corollary where I'm gonna fix gamma, I'm gonna fix my error bound, and fix delta and solve for M. And if you do that, you get the following corollary: Let H be fixed with K hypotheses and let any delta and gamma be fixed.

Then in order to guarantee that, let's say I want a guarantee that the generalization error of the hypothesis I choose with empirical risk minimization, that this is at most two times gamma worse than the best possible error I could obtain with this hypothesis class. Lets say I want this to hold true with probability at least one minus delta, then it suffices that M is [inaudible] to that. Okay? And this is sort of solving for the error bound for M. One thing we're going to convince yourselves of the easy part of this is if you set that term [inaudible] gamma and solve for M you will get this. One thing I want you to go home and sort of convince yourselves of is that this result really holds true. That this really logically follows from the theorem we've proved. In other words, you can take that formula we wrote and solve for M and – because this is the formula you get for M, that's just – that's the easy part. That once you go back and convince yourselves that this theorem is a true fact and that it does indeed logically follow from the other one. In particular, make sure that if you solve for that you really get M grading equals this, and why is this M grading that and not M less equal two, and just make sure – I can write this down and it sounds plausible why don't you just go back and convince yourself this is really true. Okay?

And it turns out that when we prove these bounds in learning theory it turns out that very often the constants are sort of loose. So it turns out that when we prove these bounds usually we're interested – usually we're not very interested in the constants, and so I write this as big O of one over gamma squared, log K over delta, and again, the key step in this is that the dependence on M with the size of the hypothesis class is logarithmic. And this will be very important later when we talk about infinite hypothesis classes. Okay? Any questions about this? No? Okay, cool. So next lecture we'll come back, we'll actually start from this result again. Remember this. I'll write this down as the first thing I do in the next lecture and we'll generalize these to infinite hypothesis classes and then talk about practical algorithms for model spectrum. So I'll see you guys in a couple days.

[End of Audio]
Duration: 75 minutes

1.10 Machine Learning Lecture 10[10]

Figure 1.10

MachineLearning-Lecture10

Instructor (Andrew Ng): So just a couple of quick announcements. One is, first, thanks again, for all of your Problem Set 1 submissions. They've all been graded, and we'll return them at the end of lecture today. If you're an SEPD student, and you submitted your Problem Set 1 by faxing it to us, then we'll return it to you via the SEPD career. And if you handed in a hard copy of your homework, instead of in

---

[10]This content is available online at <http://cnx.org/content/m45909/1.2/>.

person or in the late hand in box, then they'll be available in this classroom at the end of lecture today for you to pick up. And homework's that aren't picked up today, we'll leave at the late hand in box here in the basement of the Gates building; which, by the way, you can actually access after hours, in case any of you didn't already know that.

So you can pick them up at the end of class today or again, if you're an SEPD student, and we got your homework by fax, we'll return it to you via the SEPD career. And if you haven't seen it already, Problem Set 2 has also been posted on the web page. I think it was posted online last week. So do make sure you download that if you haven't already. I'd like to say, many of you actually did very well on Problem Set 1. As an instructor, it's actually sort of personally gratifying to me when I see your homework solutions, and I say, like, "Hey, these guys actually understood what I said," so that was actually cool. And I shall also say, thanks to all the people in this class for working as graders, and stayed up very late on Monday night to get all the grading done, so just a big thank you to the graders as well.

Let's see, one more announcement. Just a reminder, that the midterm for this class is scheduled for the 8th of November at 6:00 p.m. So the midterm will be – that's about two weeks from now, I guess. So the midterm will be open book, open notes, but please don't bring – but no laptops and computers. SEPD students, if you live in the Bay area, then I'll ask that you come in person to Stanford to take the midterm in person on the evening of the 8th of November. If you're an SEPD student, and you live outside the Bay area – so if you can't drive to Stanford to take your – then please email us at the usual class mailing address: cs229qa@cs.stanford.edu. This is the same address you see on our web pages.

If you can't physically come in to attend the midterm because you live outside the bay area, then please make sure you email us by next Wednesday, so they will make alternate arrangements for the midterm. And for regular Stanford students, as well; students that aren't taking this via SEPD, and if you have a conflict. So if you have some other event of sort of equal or greater importance than the 229 midterm, like another midterm of another class that conflicts, please also email us by next Wednesday at the usual staff mailing address to let us know; okay? And so if I don't hear from you by Wednesday, I'll assume that you'll be showing up in person for the midterm. Okay. Any questions about any of that?

Okay. So, welcome back. And before I get into this lecture's technical material, I'll just say this week's discussion section will be the TA's again talking about convex optimization. So at the last week's discussion section they discussed total convex optimization. And this week they'll wrap up the material they have to present on convex optimization.

So what I want to do today in this lecture is talk a little bit more about learning theory. In particular, I'll talk about VC dimension and building on the issues of bias variance tradeoffs of under fitting and over fitting; that we've been seeing in the previous lecture, and then we'll see in this one. I then want to talk about model selection algorithms for automatically making decisions for this bias variance tradeoff, that we started to talk about in the previous lecture. And depending on how much time, I actually may not get to Bayesian, [inaudible]. But if I don't get to this today, I'll get to this in next week's lecture.

To recap: the result we proved at the previous lecture was that if you have a finite hypothesis class – if h is a set of k hypotheses, and suppose you have some fixed parameters, gamma and delta, then in order to guarantee that this holds, we're probability at least one minus delta. It suffices that n is greater and equal to that; okay? And using big-O notations, just learning dropped constants, I can also write this as that; okay? So just to quickly remind you of what all of the notation means, we talked about empirical risk minimization, which was the simplified modern machine learning that has a hypothesis class of script h.

And what the empirical risk minimization-learning algorithm does is it just chooses the hypothesis that attains the smallest error on the training set. And so this symbol, epsilon, just denoted generalization error; right? This is the probability of a hypothesis h [inaudible] misclassifying a new example drawn from the same distribution as the training set. And so this says that in order to guarantee that the generalization error of the hypothesis h [inaudible] output by empirical risk minimization – that this is less and equal to the best possible generalization error – use it in your hypothesis class plus two times gamma – two times this error threshold. We want to guarantee that this holds a probability at least one minus delta. We show that it suffices for your training set size m to be greater than equal to this; okay? One over two gamma square log two k over delta; where again, k is the size of your hypothesis class.

And so this is some complexity result because it gives us a bound in the number of training examples we need in order to give a guarantee on something – on the error; okay? So this is a sample complexity result. So what I want to do now is take this result, and try to generalize it to the case of infinite hypothesis classes. So here, we said that the set script h is sort of just k specific functions, when you want to use a model like logistic regression, which is actually parameterized by real numbers. So I'm actually first going to give an argument that's sort of formally broken – just sort of technically somewhat broken, but conveys useful intuition. And then I'll give the more correct argument, but without proving. It's as if, full proof is somewhat involved.

So here's a somewhat broken argument. Let's say I want to apply this result analyzing logistic regression. So let's say your hypothesis class is because of all linear division boundaries; right? So say script h is parameterized by d real numbers; okay? So for example, if you're applying logistic regression with over [inaudible], then d would be endless one with logistic regression to find the linear position boundary, parameterized by endless one real numbers.

When you think about how your hypothesis class is really represented in a computer – computers use zero one bits to represent real numbers. And so if you use like a normal standard computer, it normally will represent real numbers by what's called double position floating point numbers. And what that means is that each real number is represented by or a 64-bit representation; right?

So really – you know what floating point is in a computer. So a 64-bit floating point is what almost all of us use routinely. And so this parameterized by d real numbers, that's really as if it's parameterized by 64 times d bits. Computers can't represent real numbers. They only represent – used to speed things. And so the size of your hypothesis class in your computer representation – you have 64 times d bits that you can flip. And so the number of possible values for your 62 to 64 d bits is really just to the power of 64 d; okay? Because that's the number of ways you can flip the 64 d bits. And so this is why it's important that we that we had log k there; right? So k is therefore, to the 64 d. And if I plug it into this equation over here, what you find is that in order to get this sort of guarantee, it suffices that m is great and equal to on the order of – one of the gamma square log – it's just a 64 d over delta, which is that; okay?

So just to be clear, in order to guarantee that there's only one, instead of the same complexity result as we had before – so the question is: suppose, you want a guarantee that a hypotheses returned by empirical risk minimization will have a generalization error that's within two gamma or the best hypotheses in your hypotheses class. Then what this result suggests is that, you know, in order to give that sort of error bound guarantee, it suffices that m is greater and equal to this. In other words, that your number of training examples has to be on the order of d over gamma square; 10, 12, 1 over delta. Okay? And the intuition that this conveys is actually, roughly right.

This says, that the number of training examples you need is roughly linear in the number of parameters of your hypothesis class. That m has [inaudible] on the order of something linear, [inaudible]. That intuition is actually, roughly right. I'll say more about this later. This result is clearly, slightly broken, in the sense that it relies on a 64-bit representation of 14-point numbers. So let me actually go ahead and outline the "right way" to show this more formally; all right? And it turns out the "right way" to show this more formally involves a much longer – because the proof is extremely involved, so I'm just actually going to state the result, and not prove it.

Farther proof – be a source of learning theory balance, infinite hypothesis classes. This definition – given a set of d points, we say, a hypothesis class h shatters the set s, if h can realize any labeling on it; okay? And what I mean by realizing any labeling on it – the informal way of thinking about this is: if a hypothesis class has shattered the set s, what that means is that I can take these d points, and I can associate these d points with any caught set of labels y; right? So choose any set of labeling y for each of these d points. And if your hypothesis class shatters s, then that means that there will be a hypothesis that labels those d examples perfectly; okay? That's what shattering means.

So let me just illustrate those in an example. So let's say h is the class of all linear classifiers into e, and let's say that s is this [inaudible] comprising two points; okay? So there are four possible labelings that computes with these two points. You can choose to label both positive; one positive, one negative, one negative, one positive or you can label both of them negative. And if the hypothesis class h classed all linear

classifiers into the − then, for each of these training sets, I can sort of find a linear classifier that attains zero training error on each of these. Then on all possible labelings of this set of two points. And so I'll say that the hypothesis class script h shatters this set s of two points; okay?

One more example − show you a larger example. Suppose my set s is now this set of three points; right? Then, I now have eight possible labelings for these three points; okay? And so for these three points, I now have eight possible labelings. And once again, I can − for each of these labelings, I can find the hypothesis in the hypothesis class that labels these examples correctly. And so once again, I see that − by definition, say, that my hypothesis class also shatters this set s.

Student: Right.

Instructor (Andrew Ng): And then that − that terminology − h can realize any labeling on s. That's obviously [inaudible]. Give it any set of labels and you can find a hypothesis that perfectly separates the positive and negative examples; okay? So how about this set? Suppose s is now this set of four points, then, you know, there are lots of labels. There are now 16 labelings we can choose on this; right? That's one for instance, and this is another one; right? And so I can realize some labelings. But there's no linear division boundary that can realize this labeling, and so h does not shatter this set of four points; okay? And I'm not really going to prove it here, but it turns out that you can show that in two dimensions, there is no set of four points that − the class of all linear classifiers can shatter; okay?

So here's another definition. When I say that the − well, it's called the VC dimension. These two people, Vapnik and Chervonenkis − so given a hypothesis class, the Vapnik and Chervonenkis dimension of h, which we usually write as VC of script h, is the size of the larger set that is shattered by this set − by h. And if a hypothesis class can shatter arbitrarily large sets, then the VC dimension is infinite. So just as a kind of good example: if h is the class of all linear classifiers into d, then the VC dimension of the set is equal to three because we saw just now that there is a size of − there was a set s of size three that it could shatter, and I don't really prove it. But it turns out there is no sets of size four that it can shatter. And therefore, the VC dimension of this is three. Yeah?

Student: But there are sets of size three that cannot shatter; right? [Inaudible] was your point.

Instructor (Andrew Ng): Yes, absolutely. So it turns out that if I choose a set like this − it's actually set s, then there are labelings on this they cannot realize. And so, h cannot shatter this set. But that's okay because − right − there definitely is − there exists some other set of size three being shattered. So the VC dimension is three. And then there is no set of size four that can shatter. Yeah?

Student:[Inaudible].

Instructor (Andrew Ng): Not according to this definition. No. Right. So again, let's see, I can choose my set s to be to be a set of three points that are all over lapping. Three points in exactly the same place. And clearly, I can't shatter this set, but that's okay. And I can't shatter this set, either, but that's okay because there are some other sets of size three that I can shatter. And it turns out this result holds true into the − more generally, in any dimensions − the VC dimension of the class of linear classifiers in any dimensions is equal to n plus one. Okay? So this is in [inaudible], and if you have linear classifiers over in any dimensional feature space, the VC dimension in any dimensions; whereas, n d is equal to n plus one.

So maybe you wanna write it down: what is arguably the best-known result in all of learning theory, I guess; which is that. Let a hypothesis class be given, and let the VC dimension of h be equal to d. Then we're in probability of one minus delta. We have that − the formula on the right looks a bit complicated, but don't worry about it. I'll point out the essential aspects of it later. But the key to this result is that if you have a hypothesis class with VC dimension d, and now this can be an infinite hypothesis class, what Vapnik and Chervonenkis show is that we're probability of at least one minus delta. You enjoy this sort of uniform conversions results; okay? We have that for all hypotheses h − that for all the hypotheses in your hypothesis class, you have that the generalization error of h minus the training error of h.

So the difference between these two things is bounded above by some complicated formula like this; okay? And thus, we're probably one minus delta. We also have that − have the same thing; okay? And going from this step to this step; right? Going from this step to this step is actually something that you saw yourself; that we actually proved earlier. Because − you remember, in the previous lecture we proved that if you have uniform conversions, then that implies that − it appears actually that we showed that if generalization error

and training error are close to each other; within gamma of each other, then the generalization error of the hypotheses you pick will be within two gamma times the best generalization error.

So this is really generalization error of h [inaudible] best possible generalization error plus two times gamma. And just the two constants in front here that I've absorbed into the big-O notation. So that formula is slightly more complicated. Let me just rewrite this as a corollary, which is that in order to guarantee that this holds, we're probability of one minus delta. We're probably at least one minus delta, I should say. It suffices that – I'm gonna write this – this way: I'm gonna write m equals big-O of d, and I'm going to put gamma and delta in as a subscript error to denote that. Let's see, if we treat gamma and delta as constants, so they allow me to absorb turns that depend on gamma and delta into the big-O notation, then in order to guarantee this holds, it suffices that m is on the order of the VC dimension and hypotheses class; okay?

So let's see. So what we conclude from this is that if you have a learning algorithm that tries to – for empirical risk minimization algorithms – in other words, less formally, for learning algorithms, they try to minimize training error. The intuition to take away from this is that the number of training examples you need is therefore, roughly, linear in the VC dimension of the hypotheses class. And more formally, this shows that sample complexity is upper bounded by the VC dimension; okay? It turns out that for most reasonable hypothesis classes, it turns out that the VC dimension is sort of very similar, I guess, to the number of parameters you model. So for example, you have model and logistic regression – linear classification.

In any dimensions – logistic regression in any dimensions is endless one parameters. And the VC dimension of which is the – of class of linear classifiers is always the endless one. So it turns out that for most reasonable hypothesis classes, the VC dimension is usually linear in the number of parameters of your model. Wherein, is most sense of low other polynomial; in the number of parameters of your model. And so this – the takeaway intuition from this is that the number of training examples you need to fit in those models is going to be let's say, roughly, linear in the number of parameters in your model; okay?

There are some – somewhat strange examples where what I just said is not true. There are some strange examples where you have very few parameters, but the VC dimension is enormous. But I actually know of – all of the examples I know of that fall into that regime are somewhat strange and degenerate. So somewhat unusual, and not the source of not learning algorithms you usually use.

Let's see, just other things. It turns out that – so this result shows the sample complexity is upper bounded by VC dimension. But if you have a number of training examples that are on the order of the VC dimension, then you find – it turns out that in the worse case some complexity is also lower bounded by VC dimension. And what that means is that if you have a perfectly nasty learning problem, say, then if the number of training examples you have is less than on the order of the VC dimension; then it is not possible to prove this bound. So I guess in the worse case, sample complexity in the number of training examples you need is upper bounded and lower bounded by the VC dimension.

Let's see, questions about this?

Student: Does the proof of this assume any sort of finites of, like, finite [inaudible] like you have to just [inaudible] real numbers and [inaudible]?

Instructor (Andrew Ng): Let's see. The proof is not, no. I've actually stated the entirety of the theorem. This is true. It turns out in the proof – well, somewhere, regardless of the proof there's a step reconstruction called an epsilon net, which is a very clever [inaudible]. It' sort of in regardless of the proof, it is not an assumption that you need. In someway that sort of proof – that's one-step that uses a very clever [inaudible] to prove this. But that's not needed; it's an assumption.

I'd like to say, back when I was a Ph.D. student, when I was working through this proof, there was sort of a solid week where I would wake up, and go to the office at 9:00 a.m. Then I'd start reading the book that led up to this proof. And then I'd read from 9:00 a.m. to 6:00 p.m. And then I'd go home, and then the next day, I'd pick up where I left off. And it sort of took me a whole week that way, to understand this proof, so I thought I would inflict that on you.

Just to tie a couple of loose ends: what I'm about to do is, I'm about to just mention a few things that will maybe, feel a little bit like random facts. But I'm just gonna tie up just a couple of loose ends. And so let's see, it turns out that – just so it will be more strong with you – so this bound was proved for an

algorithm that uses empirical risk minimization, for an algorithm that minimizes 0-1 training error. So one question that some of you ask is how about support vector machines; right? How come SVM's don't over fit? And in the sequel of – remember our discussion on support vector machines said that you use kernels, and map the features in infinite dimensional feature space. And so it seems like the VC dimension should be infinite; n plus one and n is infinite.

So it turns out that the class of linear separators with large margin actually has low VC dimension. I wanna say this very quickly, and informally. It's actually, not very important for you to understand the details, but I'm going to say it very informally. It turns out that I will give you a set of points. And if I ask you to consider only the course of lines that separate these points of a large margin [inaudible], so my hypothesis class will comprise only the linear position boundaries that separate the points of a large margin. Say with a margin, at least gamma; okay. And so I won't allow a point that comes closer. Like, I won't allow that line because it comes too close to one of my points.

It turns out that if I consider my data points all lie within some sphere of radius r, and if I consider only the course of linear separators is separate to data with a margin of at least gamma, then the VC dimension of this course is less than or equal to r squared over four gamma squared plus one; okay? So this funny symbol here, that just means rounding up. This is a ceiling symbol; it means rounding up x. And it turns out you prove – and there are some strange things about this result that I'm deliberately not gonna to talk about – but turns they can prove that the VC dimension of the class of linear classifiers with large margins is actually bounded. The surprising thing about this is that this is the bound on VC dimension that has no dependents on the dimension of the points x.

So in other words, your data points x combine an infinite dimensional space, but so long as you restrict attention to the class of your separators with large margin, the VC dimension is bounded. And so in trying to find a large margin separator – in trying to find the line that separates your positive and your negative examples with large margin, it turns out therefore, that the support vector machine is automatically trying to find a hypothesis class with small VC dimension. And therefore, it does not over fit. Alex?

Student: What is the [inaudible]?

Instructor (Andrew Ng): It is actually defined the same way as finite dimensional spaces. So you know, suppose you have infinite – actually, these are constantly infinite dimensional vectors; not [inaudible] to the infinite dimensional vectors. Normally, the 2 to 1 squared is equal to some [inaudible] equals 110 xi squared, so if x is infinite dimensional, you just appoint it like that. [Inaudible]. [Crosstalk]

Student:[Inaudible].

Instructor (Andrew Ng): Now, say that again.

Student:[Inaudible].

Instructor (Andrew Ng): Yes. Although, I assume that this is bounded by r.

Student: Oh.

Instructor (Andrew Ng): It's a – yeah – so this insures that conversions. So just something people sometimes wonder about. And last, the – actually – tie empirical risk minimization back a little more strongly to the source of algorithms we've talked about. It turns out that – so the theory was about, and so far, was really for empirical risk minimization. So that view's – so we focus on just one training example. Let me draw a function, you know, a zero here jumps to one, and it looks like that. And so this for once, this training example, this may be indicator h where [inaudible] is d equals data transpose x; okay? But one training example – your training example will be positive or negative. And depending on what the value of this data transpose x is, you either get it right or wrong. And so you know, I guess if your training example – if you have a positive example, then when z is positive, you get it right.

Suppose you have a negative example, so y equals 0; right? Then if z, which is data transpose x – if this is positive, then you will get this example wrong; whereas, if z is negative then you'd get this example right. And so this is a part of indicator h subscript [inaudible] x not equals y; okay? You know, it's equal to g of data transpose x; okay? And so it turns out that – so what you really like to do is choose parameters data so as to minimize this step function; right? You'd like to choose parameters data, so that you end up with a correct classification on setting your training example, and so you'd like indicator h of x not equal y. You'd like this indicator function to be 0. It turns out this step function is clearly a non-convex function.

And so it turns out that just the linear classifiers minimizing the training error is an empty heart problem. It turns out that both logistic regression, and support vector machines can be viewed as using a convex approximation for this problem. And in particular – and draw a function like that – it turns out that logistic regression is trying to maximize likelihood. And so it's tying to minimize the minus of the logged likelihood. And if you plot the minus of the logged likelihood, it actually turns out it'll be a function that looks like this. And this line that I just drew, you can think of it as a rough approximation to this step function; which is maybe what you're really trying to minimize, so you want to minimize training error.

So you can actually think of logistic regression as trying to approximate empirical risk minimization. Where instead of using this step function, which is non-convex, and gives you a hard optimization problem, it uses this line above – this curve above. So approximate it, so you have a convex optimization problem you can find the – maximum likelihood it's in the parameters for logistic regression. And it turns out, support vector machine also can be viewed as approximated dysfunction to only a little bit different – let's see, support vector machine turns out, can be viewed as trying to approximate this step function two over different approximation that's linear, and then – that sort of [inaudible] linear that – our results goes this [inaudible] there, and then it goes up as a linear function there. And that's – that is called the hinge class.

And so you can think of logistic regression and the support vector machine as different approximations to try to minimize this step function; okay? And that's why I guess, all the theory we developed – even though SVM's and logistic regression aren't exactly due to empirical risk minimization, the theory we develop often gives the completely appropriate intuitions for SVM's, and logistic regression; okay. So that was the last of the loose ends. And if you didn't get this, don't worry too much about it. It's a high-level message. It's just that SVM's and logistic regression are reasonable to think of as approximations – empirical risk minimization algorithms. What I want to do next is move on to talk about model selection. Before I do that, let me just check for questions about this. Okay. Cool.

Okay. So in the theory that we started to develop in the previous lecture, and that we sort of wrapped up with a discussion on VC dimension, we saw that there's often a trade-off between bias and variance. And in particular, so it is important not to choose a hypothesis that's either too simple or too complex. So if your data has sort of a quadratic structure to it, then if you choose a linear function to try to approximate it, then you would under fit. So you have a hypothesis with high bias. And conversely, we choose a hypothesis that's too complex, and you have high variance. And you'll also fail to fit. Then you would over fit the data, and you'd also fail to generalize well. So model selection algorithms provide a class of methods to automatically trade – make these tradeoffs between bias and variance; right?

So remember the cartoon I drew last time of generalization error? I drew this last time. Where on the x-axis was model complexity, meaning the number of – the degree of the polynomial; the [inaudible] regression function or whatever. And if you have too simple a model, you have high generalization error, those under fitting. And you if have too complex a model, like 15 or 14-degree polynomial to five data points, then you also have high generalization error, and you're over fitting. So what I wanna do now is actually just talk about model selection in the abstract; all right? Some examples of model selection problems will include – well, I'll run the example of – let's say you're trying to choose the degree of a polynomial; right? What degree polynomial do you want to choose?

Another example of a model selection problem would be if you're trying to choose the parameter [inaudible], which was the bandwidth parameter in locally awaited linear regression or in some sort of local way to regression. Yet, another model selection problem is if you're trying to choose the parameter c [inaudible] and as the [inaudible]; right? And so one known soft margin is the – we had this optimization objective; right? And the parameter c controls the tradeoff between how much you want to set for your example. So a large margin versus how much you want to penalize in this class [inaudible] example. So these are three specific examples of model selection problems.

And let's come up with a method for semantically choosing them. Let's say you have some finite set of models, and let's write these as m1, m2, m3, and so on. For example, this may be the linear classifier or this may be the quadratic classifier, and so on; okay? Or this may also be – you may also take the bandwidth parameter [inaudible] and discretize it into a range of values, and you're trying to choose from the most – discrete of the values. So let's talk about how you would select an appropriate model; all right? Well, one

thing you could do is you can pick all of these models, and train them on you're training set. And then see which model has the lowest training error. So that's a terrible idea, and why's that?

Student:[Inaudible].

Instructor (Andrew Ng): Right. Cool. Because of the over fit; right. And those – some of you are laughing that I asked that. So that'd be a terrible idea to choose a model by looking at your training set because well, obviously, you end up choosing the most complex model; right? And you choose a 10th degree polynomial because that's what fits the training set.

So we come to model selection in a training set – several standard procedures to do this. One is hold out cross validation, and in hold out cross validation, we teach a training set. And we randomly split the training set into two subsets. We call it subset – take all the data you have and randomly split it into two subsets. And we'll call it the training set, and the hold out cross validation subset. And then, you know, you train each model on just trading subset of it, and test it on your hold out cross validation set. And you pick the model with the lowest error on the hold out cross validation subset; okay?

So this is sort of a relatively straightforward procedure, and it's commonly used where you train on 70 percent of the data. Then test all of your models. And 30 percent, you can take whatever has the smallest hold out cross validation error. And after this – you actually have a chose. You can actually – having taken all of these hypothesis trained on 70 percent of the data, you can actually just output the hypothesis that has the lowest error on your hold out cross validation set. And optionally, you can actually take the model that you selected and go back, and retrain it on all 100 percent of the data; okay?

So both versions are actually done and used really often. You can either, you know, just take the best hypothesis that was trained on 70 percent of the data, and just output that as you find the hypothesis or you can use this to – say, having chosen the degree of the polynomial you want to fit, you can then go back and retrain the model on the entire 100 percent of your data. And both of these are commonly done. How about a cross validation does – sort of work straight? And sometimes we're working with a company or application or something. The many machine-learning applications we have very little data or where, you know, every training example you have was painfully acquired at great cost; right?

Sometimes your data is acquired by medical experiments, and each of these – each training example represents a sick man in amounts of physical human pain or something. So we talk and say, "Well, I'm going to hold out 30 percent of your data set, just to select my model." If people were who – sometimes that causes unhappiness, and so maybe you wanna use – not have to leave out 30 percent of your data just to do model selection.

So there are a couple of other variations on hold out cross validation that makes sometimes, slightly more efficient use of the data. And one is called k-fold cross validation. And here's the idea: I'm gonna take all of my data s; so imagine, I'm gonna draw this box s, as to note the entirety of all the data I have. And I'll then divide it into k pieces, and this is five pieces in what I've drawn. Then what'll I'll do is I will repeatedly train on k minus one pieces. Test on the remaining one – test on the remaining piece, I guess; right? And then you average over the k result.

So another way, we'll just hold out – I will hold out say, just $1/5$ of my data and I'll train on the remaining $4/5$, and I'll test on the first one. And then I'll then go and hold out the second $1/5$ from my [inaudible] for the remaining pieces – test on this, you remove the third piece, train on the $4/5$; I'm gonna do this five times. And then I'll take the five error measures I have and I'll average them. And this then gives me an estimate of the generalization error of my model; okay? And then, again, when you do k-fold cross validation, usually you then go back and retrain the model you selected on the entirety of your training set. So I drew five pieces here because that was easier for me to draw, but k equals 10 is very common; okay? I should say k equals 10 is the fairly common choice to do 10 fold cross validation.

And the advantage of the over hold out cross option is that you switch the data into ten pieces. Then each time you're only holding out $1/10$ of your data, rather than, you know, say, 30 percent of your data. I must say, in standard hold out – in simple hold out cross validation, a $30 - 70$ split is fairly common. Sometimes like $2/3 - 1/3$ or a $70 - 30$ split is fairly common. And if you use k-fold cross validation, k equals 5 or more commonly k equals 10, and is the most common choice. The disadvantage of k-fold cross validation is that it can be much more computationally expensive. In particular, to validate your model, you now need

to train your model ten times, instead of just once. And so you need to: from logistic regression, ten times per model, rather than just once. And so this is computationally more expensive. But k equals ten works great.

And then, finally, in – there's actually a version of this that you can take even further, which is when your set k equals m. And so that's when you take your training set, and you split it into as many pieces as you have training examples. And this procedure is called leave one out cross validation. And what you do is you then take out the first training example, train on the rest, and test on the first example. Then you take out the second training example, train on the rest, and test on the second example. Then you take out the third example, train on everything, but your third example. Test on the third example, and so on. And so with this many pieces you are now making, maybe even more effective use of your data than k-fold cross validation.

But you could leave – leave one out cross validation is computationally very expensive because now you need to repeatedly leave one example out, and then run your learning algorithm on m minus one training examples. You need to do this a lot of times, and so this is computationally very expensive. And typically, this is done only when you're extremely data scarce. So if you have a learning problem where you have, say, 15 training examples or something, then if you have very few training examples, leave one out cross validation is maybe preferred. Yeah?

Student: You know, that time you proved that the difference between the generalized [inaudible] by number of examples in your training set and VC dimension. So maybe [inaudible] examples into different groups, we can use that for [inaudible].

Instructor (Andrew Ng): Yeah, I mean –

Student:- compute the training error, and use that for computing [inaudible] for a generalized error.

Instructor (Andrew Ng): Yeah, that's done, but – yeah, in practice, I personally tend not to do that. It tends not to be – the VC dimension bounds are somewhat loose bounds. And so there are people – in structure risk minimization that propose what you do, but I personally tend not to do that, though. Questions for cross validation? Yeah.

Student: This is kind of far from there because when we spend all this time [inaudible] but how many data points do you sort of need to go into your certain marginal [inaudible]?

Instructor (Andrew Ng): Right.

Student: So it seems like when I'd be able to use that instead of do this; more analytically, I guess. I mean –

Instructor (Andrew Ng): Yeah.

Student:[Inaudible].

Instructor (Andrew Ng): No – okay. So it turns out that when you're proving learning theory bounds, very often the bounds will be extremely loose because you're sort of proving the worse case upper bound that holds true even for very bad – what is it – so the bounds that I proved just now; right? That holds true for absolutely any probability distribution over training examples; right? So just assume the training examples we've drawn, iid from some distribution script d, and the bounds I proved hold true for absolutely any probability distribution over script d. And chances are whatever real life distribution you get over, you know, houses and their prices or whatever, is probably not as bad as the very worse one you could've gotten; okay? And so it turns out that if you actually plug in the constants of learning theory bounds, you often get extremely large numbers.

Take logistic regression – logistic regression you have ten parameters and 0.01 error, and with 95 percent probability. How many training examples do I need? If you actually plug in actual constants into the text for learning theory bounds, you often get extremely pessimistic estimates with the number of examples you need. You end up with some ridiculously large numbers. You would need 10,000 training examples to fit ten parameters. So a good way to think of these learning theory bounds is – and this is why, also, when I write papers on learning theory bounds, I quite often use big-O notation to just absolutely just ignore the constant factors because the bounds seem to be very loose.

There are some attempts to use these bounds to give guidelines as to what model to choose, and so on. But I personally tend to use the bounds – again, intuition about – for example, what are the number of

training examples you need gross linearly in the number of parameters or what are your gross x dimension in number of parameters; whether it goes quadratic – parameters? So it's quite often the shape of the bounds. The fact that the number of training examples – the fact that some complexity is linear in the VC dimension, that's sort of a useful intuition you can get from these theories. But the actual magnitude of the bound will tend to be much looser than will hold true for a particular problem you are working on. So did that answer your question?

Student: Uh-huh.

Instructor (Andrew Ng):Yeah. And it turns out, by the way, for myself, a rule of thumb that I often use is if you're trying to fit a logistic regression model, if you have n parameters or n plus one parameters; if the number of training examples is ten times your number of parameters, then you're probably in good shape. And if your number of training examples is like tiny times the number of parameters, then you're probably perfectly fine fitting that model. So those are the sorts of intuitions that you can get from these bounds.

Student: In cross validation do we assume these examples randomly?

Instructor (Andrew Ng):Yes. So by convention we usually split the train testers randomly. One more thing I want to talk about for model selection is – there's actually a special case of model selections, called the feature selection problem. And so here's the intuition: for many machine-learning problems you may have a very high dimensional feature space with very high dimensional – you have x's – [inaudible] feature x's. So for example, for text classification – and I wanna talk about this text classification example that spam versus non-spam. You may easily have on the order of 30,000 or 50,000 features. I think I used 50,000 in my early examples. So if you have so many features – you have 50,000 features, depending on what learning algorithm you use, there may be a real risk of over fitting.

And so if you can reduce the number of features, maybe you can reduce the variance of your learning algorithm, and reduce the risk of over fitting. And for the specific case of text classification, if you imagine that maybe there's a small number of "relevant features," so there are all these English words. And many of these English words probably don't tell you anything at all about whether the email is spam or non-spam. If it were, you know, English function words like, the, of, a, and; these are probably words that don't tell you anything about whether the email is spam or non-spam. So words in contrast will be a much smaller number of features that are truly "relevant" to the learning problem.

So for example, you see the word buy or Viagra, those are words that are very useful. So you – words, some you spam and non-spam. You see the word Stanford or machine-learning or your own personal name. These are other words that are useful for telling you whether something is spam or non-spam. So in feature selection, we would like to select a subset of the features that may be or hopefully the most relevant ones for a specific learning problem, so as to give ourselves a simpler learning – a simpler hypothesis class to choose from. And then therefore, reduce the risk of over fitting. Even when we may have had 50,000 features originally.

So how do you do this? Well, if you have n features, then there are two to the n possible subsets; right? Because, you know, each of your n features can either be included or excluded. So there are two to the n possibilities. And this is a huge space. So in feature selection, what we most commonly do is use various searcheristics – sort of simple search algorithms to try to search through this space of two to the n possible subsets of features; to try to find a good subset of features. This is too large a number to enumerate all possible feature subsets.

And as a complete example, this is the forward search algorithm; it's also called the forward selection algorithm. It's actually pretty simple, but I'll just write it out. My writing it out will make it look more complicated than it really is, but it starts with – initialize the sets script f to be the empty set, and then repeat for i equals one to n; try adding feature i to the set scripts f, and evaluate the model using cross validation. And by cross validation, I mean any of the three flavors, be it simple hold out cross validation or k-fold cross validation or leave one out cross validation. And then, you know, set f to be equal to f union, I guess. And then the best feature found is f 1, I guess; okay? And finally, you would – okay?

So forward selections, procedure is: follow through the empty set of features. And then on each generation, take each of your features that isn't already in your set script f and you try adding that feature to your set.

Then you train them all, though, and evaluate them all, though, using cross validation. And basically, figure out what is the best single feature to add to your set script f. In step two here, you go ahead and add that feature to your set script f, and you get it right. And when I say best feature or best model here – by best, I really mean the best model according to hold out cross validation. By best, I really mean the single feature addition that results in the lowest hold out cross validation error or the lowest cross validation error. So you do this adding one feature at a time.

When you terminate this a little bit, as if you've added all the features to f, so f is now the entire set of features; you can terminate this. Or if by some rule of thumb, you know that you probably don't ever want more than k features, you can also terminate this if f is already exceeded some threshold number of features. So maybe if you have 100 training examples, and you're fitting logistic regression, you probably know you won't want more than 100 features. And so you stop after you have 100 features added to set f; okay? And then finally, having done this, output of best hypothesis found; again, by best, I mean, when learning this algorithm, you'd be seeing lots of hypothesis. You'd be training lots of hypothesis, and testing them using cross validation.

So when I say output best hypothesis found, I mean of all of the hypothesis you've seen during this entire procedure, pick the one with the lowest cross validation error that you saw; okay? So that's forward selection. So let's see, just to give this a name, this is an incidence of what's called wrapper feature selection. And the term wrapper comes from the fact that this feature selection algorithm that I just described is a forward selection or forward search. It's a piece of software that you write that wraps around your learning algorithm. In the sense that to perform forward selection, you need to repeatedly make cause to your learning algorithm to train your model, using different subsets of features; okay? So this is called wrapper model feature selection.

And it tends to be somewhat computationally expensive because as you're performing the search process, you're repeatedly training your learning algorithm over and over and over on all of these different subsets of features. Let's just mention also, there is a variation of this called backward search or backward selection, which is where you start with f equals the entire set of features, and you delete features one at a time; okay? So that's backward search or backward selection. And this is another feature selection algorithm that you might use. Part of whether this makes sense is really – there will be problems where it really doesn't even make sense to initialize f to be the set of all features.

So if you have 100 training examples and 10,000 features, which may well happen – 100 emails and 10,000 training – 10,000 features in email, then 100 training examples – then depending on the learning algorithm you're using, it may or may not make sense to initialize the set f to be all features, and train them all by using all features. And if it doesn't make sense, then you can train them all by using all features; then forward selection would be more common.

So let's see. Wrapper model feature selection algorithms tend to work well. And in particular, they actually often work better than a different class of algorithms I'm gonna talk about now. But their main disadvantage is that they're computationally very expensive. Do you have any questions about this before I talk about the other? Yeah?

Student:[Inaudible].

Instructor (Andrew Ng): Yeah – yes, you're actually right. So forward search and backward search, both of these are searcheristics, and you cannot – but for either of these you cannot guarantee they'll find the best subset of features. It actually turns out that under many formulizations of the feature selection problems – it actually turns out to be an empty heart problem, to find the best subset of features. But in practice, forward selection backward selection work fine, and you can also envision other search algorithms where you sort of have other methods to search through the space up to the end possible feature subsets.

So let's see. Wrapper feature selection tends to work well when you can afford to do it computationally. But for problems such as text classification – it turns out for text classification specifically because you have so many features, and easily have 50,000 features. Forward selection would be very, very expensive. So there's a different class of algorithms that will give you – that tends not to do as well in the sense of generalization error. So you tend to learn the hypothesis that works less well, but is computationally much less expensive. And these are called the filter feature selection methods. And the basic idea is that for each

feature i will compute some measure of how informative xi is about y; okay?

And to do this, we'll use some simple heuristics; for every feature we'll just try to compute some rough estimate or compute some measure of how informative xi is about y. So there are many ways you can do this. One way you can choose is to just compute the correlation between xi and y. And just for each of your features just see how correlated this is with your class label y. And then you just pick the top k most correlated features. Another way to do this − for the case of text classification, there's one other method, which especially for this k features I guess − there's one other informative measure that's used very commonly, which is called major information.

I'm going to tell you some of these ideas in problem sets, but I'll just say this very briefly. So the major information between feature xi and y − I'll just write out the definition, I guess. Let's say this is text classification, so x can take on two values, 0, 1; the major information between xi and y is to find out some overall possible values of x; some overall possible values of y times the distribution times that. Where all of these distributions − where so the joint distribution over xi and y, you would estimate from your training data all of these things you would use, as well. You would estimate from the training data what is the probability that x is 0, what's the probability that x is one, what's the probability that x is 0, y is 0, x is one; y is 0, and so on.

So it turns out there's a standard information theoretic measure of how different probability distributions are. And I'm not gonna prove this here. But it turns out that this major information is actually − so the standard measure of how different distributions are; called the K-L divergence. When you take a class in information theory, you have seen concepts of mutual information in the K-L divergence, but if you haven't, don't worry about it. Just the intuition is there's something called K-L divergence that's a formal measure of how different two probability distributions are. And mutual information is a measure for how different − the joint distribution is of x and y; from the distribution you would get − if you were to assume they were independent; okay?

So if x and y were independent, then p of x, y would be equal to p of x times p of y. And so you know, this distribution and this distribution would be identical, and the K-L divergence would be 0. In contrast, if x and y were very non-independent − in other words, if x and y are very informative about each other, then this K-L divergence will be large. And so mutual information is a formal measure of how non-independent x and y are. And if x and y are highly non-independent then that means that x will presumably tell you something about y, and so they'll have large mutual information. And this measure of information will tell you x might be a good feature. And you get to play with some of these ideas more in the problem sets. So I won't say much more about it.

And what you do then is − having chosen some measure like correlation or major information or something else, you then pick the top k features; meaning that you compute correlation between xi and y for all the features of mutual information − xi and y for all the features. And then you include in your learning algorithm the k features of the largest correlation with the label or the largest mutual information label, whatever. And to choose k, you can actually use cross validation, as well; okay? So you would take all your features, and sort them in decreasing order of mutual information. And then you'd try using just the top one feature, the top two features, the top three features, and so on. And you decide how many features includes using cross validation; okay? Or you can − sometimes you can just choose this by hand, as well.

Okay. Questions about this? Okay. Cool. Great. So next lecture I'll continue − I'll wrap up the Bayesian model selection, but less close to the end.

[End of Audio]

Duration: 77 minutes

1.11 Machine Learning Lecture 11[11]

Figure 1.11

MachineLearning-Lecture11

Instructor (Andrew Ng): Okay. Good morning. Welcome back.

What I want to do today is actually wrap up our discussion on learning theory and sort of on – and I'm gonna start by talking about Bayesian statistics and regularization, and then take a very brief digression to tell you about online learning. And most of today's lecture will actually be on various pieces of that, so applying machine learning algorithms to problems like, you know, like the project or other problems you may go work on after you graduate from this class.

But let's start the talk about Bayesian statistics and regularization. So you remember from last week, we started to talk about learning theory and we learned about bias and variance. And I guess in the previous lecture, we spent most of the previous lecture talking about algorithms for model selection and for feature selection. We talked about cross-validation. Right?

So most of the methods we talked about in the previous lecture were ways for you to try to simply the model. So for example, the feature selection algorithms we talked about gives you a way to eliminate a number of features, so as to reduce the number of parameters you need to fit and thereby reduce overfitting. Right? You remember that? So feature selection algorithms choose a subset of the features so that you have less parameters and you may be less likely to overfit. Right?

What I want to do today is to talk about a different way to prevent overfitting. And there's a method called regularization and there's a way that lets you keep all the parameters. So here's the idea, and I'm gonna illustrate this example with, say, linear regression. So you take the linear regression model, the very first model we learned about, right, we said that we would choose the parameters via maximum likelihood. Right? And that meant that, you know, you would choose the parameters theta that maximized the probability of the data, which is parameters theta that maximized the probability of the data we observe. Right?

And so to give this sort of procedure a name, this is one example of most common frequencies procedure, and frequency, you can think of sort of as maybe one school of statistics. And the philosophical view behind writing this down was we envisioned that there was some true parameter theta out there that generated, you know, the Xs and the Ys. There's some true parameter theta that govern housing prices, Y is a function of X, and we don't know what the value of theta is, and we'd like to come up with some procedure for estimating the value of theta. Okay? And so, maximum likelihood is just one possible procedure for estimating the unknown value for theta.

And the way you formulated this, you know, theta was not a random variable. Right? That's what why said, so theta is just some true value out there. It's not random or anything, we just don't know what it is, and we have a procedure called maximum likelihood for estimating the value for theta. So this is one example of what's called a frequencies procedure.

The alternative to the, I guess, the frequency school of statistics is the Bayesian school, in which we're gonna say that we don't know what theta, and so we will put a prior on theta. Okay? So in the Bayesian school students would say, "Well don't know what the value of theta so let's represent our uncertainty over theta with a prior." So for example, our prior on theta may be a Gaussian distribution with mean zero and curvalence matrix given by tau squared I. Okay?

And so – actually, if I use S to denote my training set, well – right, so theta represents my beliefs about what the parameters are in the absence of any data. So not having seen any data, theta represents, you

---

[11]This content is available online at <http://cnx.org/content/m45918/1.2/>.

know, what I think theta – it probably represents what I think theta is most likely to be. And so given the training set, S, in the sort of Bayesian procedure, we would, well, calculate the probability, the posterior probability by parameters given my training sets, and – let's write this on the next board.

So my posterior on my parameters given my training set, by Bayes' rule, this will be proportional to, you know, this. Right? So by Bayes' rule. Let's call it posterior. And this distribution now represents my beliefs about what theta is after I've seen the training set.

And when you now want to make a new prediction on the price of a new house, on the input X, I would say that, well, the distribution over the possible housing prices for this new house I'm trying to estimate the price of, say, given the size of the house, the features of the house at X, and the training set I had previously, it is going to be given by an integral over my parameters theta of probably of Y given X comma theta and times the posterior distribution of theta given the training set. Okay?

And in particular, if you want your prediction to be the expected value of Y given the input X in training set, you would say integrate over Y times the posterior. Okay? You would take an expectation of Y with respect to your posterior distribution. Okay?

And you notice that when I was writing this down, so with the Bayesian formulation, and now started to write up here Y given X comma theta because this formula now is the property of Y conditioned on the values of the random variables X and theta. So I'm no longer writing semicolon theta, I'm writing comma theta because I'm now treating theta as a random variable.

So all of this is somewhat abstract but this is – and it turns out – actually let's check. Are there questions about this? No? Okay.

Let's try to make this more concrete. It turns out that for many problems, both of these steps in the computation are difficult because if, you know, theta is an N plus one-dimensional vector, is an N plus one-dimensional parameter vector, then this is one an integral over an N plus one-dimensional, you know, over RN plus one. And because numerically it's very difficult to compute integrals over very high dimensional spaces, all right?

So usually this integral – actually usually it's hard to compute the posterior in theta and it's also hard to compute this integral if theta is very high dimensional. There are few exceptions for which this can be done in closed form, but for many learning algorithms, say, Bayesian logistic regression, this is hard to do.

And so what's commonly done is to take the posterior distribution and instead of actually computing a full posterior distribution, chi of theta given S, we'll instead take this quantity on the right-hand side and just maximize this quantity on the right-hand side. So let me write this down. So commonly, instead of computing the full posterior distribution, we will choose the following. Okay?

We will choose what's called the MAP estimate, or the maximum a posteriori estimate of theta, which is the most likely value of theta, most probable value of theta onto your posterior distribution. And that's just ont max chi of theta. And then when you need to make a prediction, you know, you would just predict, say, well, using your usual hypothesis and using this MAP value of theta in place of – as the parameter vector you'd choose. Okay?

And notice, the only difference between this and standard maximum likelihood estimation is that when you're choosing, you know, the – instead of choosing the maximum likelihood value for theta, you're instead maximizing this, which is what you have for maximum likelihood estimation, and then times this other quantity which is the prior. Right?

And let's see, when intuition is that if your prior is theta being Gaussian and with mean zero and some covariance, then for a distribution like this, most of the [inaudible] mass is close to zero. Right? So there's a Gaussian centered around the point zero, and so [inaudible] mass is close to zero. And so the prior distribution, instead of saying that you think most of the parameters should be close to zero, and if you remember our discussion on feature selection, if you eliminate a feature from consideration that's the same as setting the source and value of theta to be equal to zero.

All right? So if you set theta five to be equal to zero, that's the same as, you know, eliminating feature five from the your hypothesis. And so, this is the prior that drives most of the parameter values to zero – to values close to zero. And you'll think of this as doing something analogous, if – doing something reminiscent of feature selection. Okay? And it turns out that with this formulation, the parameters won't actually be

exactly zero but many of the values will be close to zero.

And I guess in pictures, if you remember, I said that if you have, say, five data points and you fit a fourth-order polynomial – well I think that had too many bumps in it, but never mind. If you fit it a – if you fit very high polynomial to a very small dataset, you can get these very large oscillations if you use maximum likelihood estimation. All right?

In contrast, if you apply this sort of Bayesian regularization, you can actually fit a higher-order polynomial that still get sort of a smoother and smoother fit to the data as you decrease tau. So as you decrease tau, you're driving the parameters to be closer and closer to zero. And that in practice – it's sort of hard to see, but you can take my word for it. As tau becomes smaller and smaller, the curves you tend to fit your data also become smoother and smoother, and so you tend less and less overfit, even when you're fitting a large number of parameters. Okay?

Let's see, and one last piece of intuition that I would just toss out there. And you get to play more with this particular set of ideas more in Problem Set 3, which I'll post online later this week I guess. Is that whereas maximum likelihood tries to minimize, say, this, right?

Whereas maximum likelihood for, say, linear regression turns out to be minimizing this, it turns out that if you add this prior term there, it turns out that the authorization objective you end up optimizing turns out to be that. Where you add an extra term that, you know, penalizes your parameter theta as being large. And so this ends up being an algorithm that's very similar to maximum likelihood, expect that you tend to keep your parameters small. And this has the effect.

Again, it's kind of hard to see but just take my word for it. That strengthening the parameters has the effect of keeping the functions you fit to be smoother and less likely to overfit. Okay? Okay, hopefully this will make more sense when you play with these ideas a bit more in the next problem set. But let's check questions about all this.

Student: The smoothing behavior is it because [inaudible] actually get different [inaudible]?

Instructor (Andrew Ng):Let's see. Yeah. It depends on – well most priors with most of the mass close to zero will get this effect, I guess. And just by convention, the Gaussian prior is what's most – used the most common for models like logistic regression and linear regression, generalized in your models. There are a few other priors that I sometimes use, like the Laplace prior, but all of them will tend to have these sorts of smoothing effects.

All right. Cool.

And so it turns out that for problems like text classification, text classification is like 30,000 features or 50,000 features, where it seems like an algorithm like logistic regression would be very much prone to overfitting. Right? So imagine trying to build a spam classifier, maybe you have 100 training examples but you have 30,000 features or 50,000 features, that seems clearly to be prone to overfitting. Right? But it turns out that with this sort of Bayesian regularization, with [inaudible] Gaussian, logistic regression becomes a very effective text classification algorithm with this sort of Bayesian regularization.

Alex?

Student:[Inaudible]?

Instructor (Andrew Ng):Yeah, right, and so pick – and to pick either tau squared or lambda. I think the relation is lambda equals one over tau squared. But right, so pick either tau squared or lambda, you could use cross-validation, yeah. All right? Okay, cool.

So all right, that was all I want to say about methods for preventing overfitting. What I want to do next is just spend, you know, five minutes talking about online learning. And this is sort of a digression. And so, you know, when you're designing the syllabus of a class, I guess, sometimes there are just some ideas you want to talk about but can't find a very good place to fit in anywhere. So this is one of those ideas that may seem a bit disjointed from the rest of the class but I just want to tell you a little bit about it.

Okay. So here's the idea. So far, all the learning algorithms we've talked about are what's called batch learning algorithms, where you're given a training set and then you get to run your learning algorithm on the training set and then maybe you test it on some other test set. And there's another learning setting called online learning, in which you have to make predictions even while you are in the process of learning. So here's how the problem sees. All right?

I'm first gonna give you X one. Let's say there's a classification problem, so I'm first gonna give you X one and then gonna ask you, you know, "Can you make a prediction on X one? Is the label one or zero?" And you've not seen any data yet. And so, you make a guess. Right? You guess – we'll call your guess Y hat one. And after you've made your prediction, I will then reveal to you the true label Y one. Okay? And not having seen any data before, your odds of getting the first one right are only 50 percent, right, if you guess randomly.

And then I show you X two. And then I ask you, "Can you make a prediction on X two?" And so you now maybe are gonna make a slightly more educated guess and call that Y hat two. And after you've made your guess, I reveal the true label to you. And so, then I show you X three, and then you make your guess, and learning proceeds as follows.

So this is just a lot of machine learning and batch learning, and the model settings where you have to keep learning even as you're making predictions, okay? So I don't know, setting your website and you have users coming in. And as the first user comes in, you need to start making predictions already about what the user likes or dislikes. And there's only, you know, as you're making predictions you get to show more and more training examples.

So in online learning what you care about is the total online error, which is sum from I equals one to MC if you get the sequence of M examples all together, indicator Y hat I not equal to Y hi. Okay? So the total online error is the total number of mistakes you make on a sequence of examples like this.

And it turns out that, you know, many of the learning algorithms you have – when you finish all the learning algorithms, you've learned about and can apply to this setting. One thing you could do is when you're asked to make prediction on Y hat three, right, one simple thing to do is well you've seen some other training examples up to this point so you can just take your learning algorithm and run it on the examples, you know, leading up to Y hat three. So just run the learning algorithm on all the examples you've seen previous to being asked to make a prediction on certain example, and then use your learning algorithm to make a prediction on the next example.

And it turns out that there are also algorithms, especially the algorithms that we saw that you could use the stochastic gradient descent, that, you know, can be adapted very nicely to this. So as a concrete example, if you remember the perceptron algorithms, say, right, you would say initial the parameter theta to be equal to zero.

And then after seeing the Ith training example, you'd update the parameters, you know, using – you've see this reel a lot of times now, right, using the standard perceptron learning rule. And the same thing, if you were using logistic regression you can then, again, after seeing each training example, just run, you know, essentially run one-step stochastic gradient descent on just the example you saw. Okay?

And so the reason I've put this into the sort of "learning theory" section of this class was because it turns that sometimes you can prove fairly amazing results on your total online error using algorithms like these. I will actually – I don't actually want to spend the time in the main lecture to prove this, but, for example, you can prove that when you use the perceptron algorithm, then even when the features XI, maybe infinite dimensional feature vectors, like we saw for simple vector machines. And sometimes, infinite feature dimensional vectors may use kernel representations. Okay?

But so it turns out that you can prove that when you a perceptron algorithm, even when the data is maybe extremely high dimensional and it seems like you'd be prone to overfitting, right, you can prove that so as the long as the positive and negative examples are separated by a margin, right.

So in this infinite dimensional space, so long as, you know, there is some margin down there separating the positive and negative examples, you can prove that perceptron algorithm will converge to a hypothesis that perfectly separates the positive and negative examples. Okay? And then so after seeing only a finite number of examples, it'll converge to digital boundary that perfectly separates the positive and negative examples, even though you may in an infinite dimensional space. Okay?

So let's see. The proof itself would take me sort of almost an entire lecture to do, and there are sort of other things that I want to do more than that. So you want to see the proof of this yourself, it's actually written up in the lecture notes that I posted online. For the purposes of this class' syllabus, the proof of this result, you can treat this as optional reading. And by that, I mean, you know, it won't appear on the

midterm and you won't be asked about this specifically in the problem sets, but I thought it'd be –

I know some of you are curious after the previous lecture so why you can prove that, you know, SVMs can have bounded VC dimension, even in these infinite dimensional spaces, and how do you prove things in these – how do you prove learning theory results in these infinite dimensional feature spaces. And so the perceptron bound that I just talked about was the simplest instance I know of that you can sort of read in like half an hour and understand it.

So if you're interested, there are lecture notes online for how this perceptron bound is actually proved. It's a very [inaudible], you can prove it in like a page or so, so go ahead and take a look at that if you're interested. Okay?

But regardless of the theoretical results, you know, the online learning setting is something that you – that comes reasonably often. And so, these algorithms based on stochastic gradient descent often go very well. Okay, any questions about this before I move on? All right. Cool.

So the last thing I want to do today, and was the majority of today's lecture, actually can I switch to PowerPoint slides, please, is I actually want to spend most of today's lecture sort of talking about advice for applying different machine learning algorithms.

And so, you know, right now, already you have a, I think, a good understanding of really the most powerful tools known to humankind in machine learning. Right? And what I want to do today is give you some advice on how to apply them really powerfully because, you know, the same tool – it turns out that you can take the same machine learning tool, say logistic regression, and you can ask two different people to apply it to the same problem. And sometimes one person will do an amazing job and it'll work amazingly well, and the second person will sort of not really get it to work, even though it was exactly the same algorithm. Right?

And so what I want to do today, in the rest of the time I have today, is try to convey to you, you know, some of the methods for how to make sure you're one of – you really know how to get these learning algorithms to work well in problems. So just some caveats on what I'm gonna, I guess, talk about in the rest of today's lecture.

Something I want to talk about is actually not very mathematical but is also some of the hardest, most conceptually most difficult material in this class to understand. All right? So this is not mathematical but this is not easy. And I want to say this caveat some of what I'll say today is debatable. I think most good machine learning people will agree with most of what I say but maybe not everything I say. And some of what I'll say is also not good advice for doing machine learning either, so I'll say more about this later.

What I'm focusing on today is advice for how to just get stuff to work. If you work in the company and you want to deliver a product or you're, you know, building a system and you just want your machine learning system to work. Okay? Some of what I'm about to say today isn't great advice if you goal is to invent a new machine learning algorithm, but this is advice for how to make machine learning algorithm work and, you know, and deploy a working system.

So three key areas I'm gonna talk about. One: diagnostics for debugging learning algorithms. Second: sort of talk briefly about error analyses and ablative analysis. And third, I want to talk about just advice for how to get started on a machine-learning problem. And one theme that'll come up later is it turns out you've heard about premature optimization, right, in writing software. This is when someone over-designs from the start, when someone, you know, is writing piece of code and they choose a subroutine to optimize heavily. And maybe you write the subroutine as assembly or something.

And that's often – and many of us have been guilty of premature optimization, where we're trying to get a piece of code to run faster. And we choose probably a piece of code and we implement it an assembly, and really tune and get to run really quickly. And it turns out that wasn't the bottleneck in the code at all. Right? And we call that premature optimization. And in undergraduate programming classes, we warn people all the time not to do premature optimization and people still do it all the time. Right?

And turns out, a very similar thing happens in building machine-learning systems. That many people are often guilty of, what I call, premature statistical optimization, where they heavily optimize part of a machine learning system and that turns out not to be the important piece. Okay? So I'll talk about that later, as well.

So let's first talk about debugging learning algorithms. As a motivating example, let's say you want to build an anti-spam system. And let's say you've carefully chosen, you know, a small set of 100 words to use as features. All right? So instead of using 50,000 words, you're chosen a small set of 100 features to use for your anti-spam system. And let's say you implement Bayesian logistic regression, implement gradient descent, and you get 20 percent test error, which is unacceptably high. Right?

So this is Bayesian logistic regression, and so it's just like maximum likelihood but, you know, with that additional lambda squared term. And we're maximizing rather than minimizing as well, so there's a minus lambda theta square instead of plus lambda theta squared. So the question is, you implemented your Bayesian logistic regression algorithm, and you tested it on your test set and you got unacceptably high error, so what do you do next? Right?

So, you know, one thing you could do is think about the ways you could improve this algorithm. And this is probably what most people will do instead of, "Well let's sit down and think what could've gone wrong, and then we'll try to improve the algorithm." Well obviously having more training data could only help, so one thing you can do is try to get more training examples. Maybe you suspect, that even 100 features was too many, so you might try to get a smaller set of features.

What's more common is you might suspect your features aren't good enough, so you might spend some time, look at the email headers, see if you can figure out better features for, you know, finding spam emails or whatever. Right. And right, so and just sit around and come up with better features, such as for email headers. You may also suspect that gradient descent hasn't quite converged yet, and so let's try running gradient descent a bit longer to see if that works. And clearly, that can't hurt, right, just run gradient descent longer.

Or maybe you remember, you know, you remember hearing from class that maybe Newton's method converges better, so let's try that instead. You may want to tune the value for lambda, because not sure if that was the right thing, or maybe you even want to an SVM because maybe you think an SVM might work better than logistic regression.

So I only listed eight things here, but you can imagine if you were actually sitting down, building machine-learning system, the options to you are endless. You can think of, you know, hundreds of ways to improve a learning system. And some of these things like, well getting more training examples, surely that's gonna help, so that seems like it's a good use of your time. Right?

And it turns out that this [inaudible] of picking ways to improve the learning algorithm and picking one and going for it, it might work in the sense that it may eventually get you to a working system, but often it's very time-consuming. And I think it's often a largely – largely a matter of luck, whether you end up fixing what the problem is.

In particular, these eight improvements all fix very different problems. And some of them will be fixing problems that you don't have. And if you can rule out six of eight of these, say, you could – if by somehow looking at the problem more deeply, you can figure out which one of these eight things is actually the right thing to do, you can save yourself a lot of time. So let's see how we can go about doing that.

The people in industry and in research that I see that are really good, would not go and try to change a learning algorithm randomly. There are lots of things that obviously improve your learning algorithm, but the problem is there are so many of them it's hard to know what to do. So you find all the really good ones that run various diagnostics to figure out the problem is and they think where a problem is. Okay?

So for our motivating story, right, we said – let's say Bayesian logistic regression test error was 20 percent, which let's say is unacceptably high. And let's suppose you suspected the problem is either overfitting, so it's high bias, or you suspect that, you know, maybe you have two few features that classify as spam, so there's –

Oh excuse me; I think I wrote that wrong. Let's firstly – so let's forget – forget the tables.

Suppose you suspect the problem is either high bias or high variance, and some of the text here doesn't make sense. And you want to know if you're overfitting, which would be high variance, or you have too few features classified as spam, it'd be high bias. I had those two reversed, sorry. Okay?

So how can you figure out whether the problem is one of high bias or high variance? Right? So it turns out there's a simple diagnostic you can look at that will tell you whether the problem is high bias or high

variance. If you remember the cartoon we'd seen previously for high variance problems, when you have high variance the training error will be much lower than the test error. All right?

When you have a high variance problem, that's when you're fitting your training set very well. That's when you're fitting, you know, a tenth order polynomial to 11 data points. All right? And that's when you're just fitting the data set very well, and so your training error will be much lower than your test error.

And in contrast, if you have high bias, that's when your training error will also be high. Right? That's when your data is quadratic, say, but you're fitting a linear function to it and so you aren't even fitting your training set well.

So just in cartoons, I guess, this is a – this is what a typical learning curve for high variance looks like. On your horizontal axis, I'm plotting the training set size M, right, and on vertical axis, I'm plotting the error. And so, let's see, you know, as you increase – if you have a high variance problem, you'll notice as the training set size, M, increases, your test set error will keep on decreasing. And so this sort of suggests that, well, if you can increase the training set size even further, maybe if you extrapolate the green curve out, maybe that test set error will decrease even further. All right?

Another thing that's useful to plot here is – let's say the red horizontal line is the desired performance you're trying to reach, another useful thing to plot is actually the training error. Right? And it turns out that your training error will actually grow as a function of the training set size because the larger your training set, the harder it is to fit, you know, your training set perfectly. Right?

So this is just a cartoon, don't take it too seriously, but in general, your training error will actually grow as a function of your training set size. Because smart training sets, if you have one data point, it's really easy to fit that perfectly, but if you have 10,000 data points, it's much harder to fit that perfectly. All right?

And so another diagnostic for high variance, and the one that I tend to use more, is to just look at training versus test error. And if there's a large gap between them, then this suggests that, you know, getting more training data may allow you to help close that gap. Okay? So this is what the cartoon would look like when – in the case of high variance.

This is what the cartoon looks like for high bias. Right? If you look at the learning curve, you see that the curve for test error has flattened out already. And so this is a sign that, you know, if you get more training examples, if you extrapolate this curve further to the right, it's maybe not likely to go down much further. And this is a property of high bias: that getting more training data won't necessarily help.

But again, to me the more useful diagnostic is if you plot training errors well, if you look at your training error as well as your, you know, hold out test set error. If you find that even your training error is high, then that's a sign that getting more training data is not going to help. Right?

In fact, you know, think about it, training error grows as a function of your training set size. And so if your training error is already above your level of desired performance, then getting even more training data is not going to reduce your training error down to the desired level of performance. Right? Because, you know, your training error sort of only gets worse as you get more and more training examples. So if you extrapolate further to the right, it's not like this blue line will come back down to the level of desired performance. Right? This will stay up there. Okay?

So for me personally, I actually, when looking at a curve like the green curve on test error, I actually personally tend to find it very difficult to tell if the curve is still going down or if it's [inaudible]. Sometimes you can tell, but very often, it's somewhat ambiguous. So for me personally, the diagnostic I tend to use the most often to tell if I have a bias problem or a variance problem is to look at training and test error and see if they're very close together or if they're relatively far apart. Okay?

And so, going back to the list of fixes, look at the first fix, getting more training examples is a way to fix high variance. Right? If you have a high variance problem, getting more training examples will help. Trying a smaller set of features: that also fixes high variance. All right? Trying a larger set of features or adding email features, these are solutions that fix high bias. Right? So high bias being if you're hypothesis was too simple, you didn't have enough features. Okay?

And so quite often you see people working on machine learning problems and they'll remember that getting more training examples helps. And so, they'll build a learning system, build an anti-spam system and it doesn't work. And then they go off and spend lots of time and money and effort collecting more

training data because they'll say, "Oh well, getting more data's obviously got to help." But if they had a high bias problem in the first place, and not a high variance problem, it's entirely possible to spend three months or six months collecting more and more training data, not realizing that it couldn't possibly help. Right?

And so, this actually happens a lot in, you know, in Silicon Valley and companies, this happens a lot. There will often people building various machine learning systems, and they'll often – you often see people spending six months working on fixing a learning algorithm and you could've told them six months ago that, you know, that couldn't possibly have helped. But because they didn't know what the problem was, and they'd easily spend six months trying to invent new features or something.

And this is – you see this surprisingly often and this is somewhat depressing. You could've gone to them and told them, "I could've told you six months ago that this was not going to help." And the six months is not a joke, you actually see this. And in contrast, if you actually figure out the problem's one of high bias or high variance, then you can rule out two of these solutions and save yourself many months of fruitless effort. Okay?

I actually want to talk about these four at the bottom as well. But before I move on, let me just check if there were questions about what I've talked about so far. No? Okay, great.

So bias versus variance is one thing that comes up often. This bias versus variance is one common diagnostic. And so, for other machine learning problems, it's often up to your own ingenuity to figure out your own diagnostics to figure out what's wrong. All right? So if a machine-learning algorithm isn't working, very often it's up to you to figure out, you know, to construct your own tests. Like do you look at the difference training and test errors or do you look at something else? It's often up to your own ingenuity to construct your own diagnostics to figure out what's going on.

What I want to do is go through another example. All right? And this one is slightly more contrived but it'll illustrate another common question that comes up, another one of the most common issues that comes up in applying learning algorithms.

So in this example, it's slightly more contrived, let's say you implement Bayesian logistic regression and you get 2 percent error on spam mail and 2 percent error non-spam mail. Right? So it's rejecting, you know, 2 percent of – it's rejecting 98 percent of your spam mail, which is fine, so 2 percent of all spam gets through which is fine, but is also rejecting 2 percent of your good email, 2 percent of the email from your friends and that's unacceptably high, let's say.

And let's say that a simple vector machine using a linear kernel gets 10 percent error on spam and 0.01 percent error on non-spam, which is more of the acceptable performance you want. And let's say for the sake of this example, let's say you're trying to build an anti-spam system. Right? Let's say that you really want to deploy logistic regression to your customers because of computational efficiency or because you need retrain overnight every day, and because logistic regression just runs more easily and more quickly or something. Okay?

So let's say you want to deploy logistic regression, but it's just not working out well. So question is: What do you do next? So it turns out that this – the issue that comes up here, the one other common question that comes up is a question of is the algorithm converging. So you might suspect that maybe the problem with logistic regression is that it's just not converging. Maybe you need to run iterations.

And it turns out that, again if you look at the optimization objective, say, logistic regression is, let's say, optimizing J of theta, it actually turns out that if you look at optimizing your objective as a function of the number of iterations, when you look at this curve, you know, it sort of looks like it's going up but it sort of looks like there's absentiles.

And when you look at these curves, it's often very hard to tell if the curve has already flattened out. All right? And you look at these curves a lot so you can ask: Well has the algorithm converged? When you look at the J of theta like this, it's often hard to tell. You can run this ten times as long and see if it's flattened out. And you can run this ten times as long and it'll often still look like maybe it's going up very slowly, or something. Right? So a better diagnostic for what logistic regression is converged than looking at this curve.

The other question you might wonder – the other thing you might suspect is a problem is are you

optimizing the right function. So what you care about, right, in spam, say, is a weighted accuracy function like that. So A of theta is, you know, sum over your examples of some weights times whether you got it right. And so the weight may be higher for non-spam than for spam mail because you care about getting your predictions correct for spam email much more than non-spam mail, say.

So let's say A of theta is the optimization objective that you really care about, but Bayesian logistic regression is that it optimizes a quantity like that. Right? It's this sort of maximum likelihood thing and then with this two-nom, you know, penalty thing that we saw previously. And you might be wondering: Is this the right optimization function to be optimizing. Okay? And: Or do I maybe need to change the value for lambda to change this parameter? Or: Should I maybe really be switching to support vector machine optimization objective?

Okay? Does that make sense?

So the second diagnostic I'm gonna talk about is let's say you want to figure out is the algorithm converging, is the optimization algorithm converging, or is the problem with the optimization objective I chose in the first place? Okay?

So here's the diagnostic you can use. Let me let – right. So to just reiterate the story, right, let's say an SVM outperforms Bayesian logistic regression but you really want to deploy Bayesian logistic regression to your problem. Let me let theta subscript SVM, be the parameters learned by an SVM, and I'll let theta subscript BLR be the parameters learned by Bayesian logistic regression.

So the optimization objective you care about is this, you know, weighted accuracy criteria that I talked about just now. And the support vector machine outperforms Bayesian logistic regression. And so, you know, the weighted accuracy on the support-vector-machine parameters is better than the weighted accuracy for Bayesian logistic regression.

So further, Bayesian logistic regression tries to optimize an optimization objective like that, which I denoted J theta. And so, the diagnostic I choose to use is to see if J of SVM is bigger-than or less-than J of BLR. Okay? So I explain this on the next slide.

So we know two facts. We know that – well we know one fact. We know that a weighted accuracy of support vector machine, right, is bigger than this weighted accuracy of Bayesian logistic regression. So in order for me to figure out whether Bayesian logistic regression is converging, or whether I'm just optimizing the wrong objective function, the diagnostic I'm gonna use and I'm gonna check if this equality hold through. Okay?

So let me explain this, so in Case 1, right, it's just those two equations copied over. In Case 1, let's say that J of SVM is, indeed, is greater than J of BLR – or J of theta SVM is greater than J of theta BLR. But we know that Bayesian logistic regression was trying to maximize J of theta; that's the definition of Bayesian logistic regression.

So this means that theta – the value of theta output that Bayesian logistic regression actually fails to maximize J because the support back to machine actually returned the value of theta that, you know does a better job out-maximizing J. And so, this tells me that Bayesian logistic regression didn't actually maximize J correctly, and so the problem is with the optimization algorithm. The optimization algorithm hasn't converged.

The other case is as follows, where J of theta SVM is less-than/equal to J of theta BLR. Okay? In this case, what does that mean? This means that Bayesian logistic regression actually attains the higher value for the optimization objective J then doesn't support back to machine. The support back to machine, which does worse on your optimization problem, actually does better on the weighted accuracy measure.

So what this means is that something that does worse on your optimization objective, on J, can actually do better on the weighted accuracy objective. And this really means that maximizing J of theta, you know, doesn't really correspond that well to maximizing your weighted accuracy criteria. And therefore, this tells you that J of theta is maybe the wrong optimization objective to be maximizing. Right? That just maximizing J of theta just wasn't a good objective to be choosing if you care about the weighted accuracy. Okay?

Can you raise your hand if this made sense? Cool, good.

So that tells us whether the problem is with the optimization objective or whether it's with the objective

function. And so going back to this slide, the eight fixes we had, you notice that if you run gradient descent for more iterations that fixes the optimization algorithm. You try and use this method fixes the optimization algorithm, whereas using a different value for lambda, in that lambda times norm of data squared, you know, in your objective, fixes the optimization objective. And changing to an SVM is also another way of trying to fix the optimization objective. Okay?

And so once again, you actually see this quite often that – actually, you see it very often, people will have a problem with the optimization objective and be working harder and harder to fix the optimization algorithm. That's another very common pattern that the problem is in the formula from your J of theta, that often you see people, you know, just running more and more iterations of gradient descent. Like trying Newton's method and trying conjugate and then trying more and more crazy optimization algorithms, whereas the problem was, you know, optimizing J of theta wasn't going to fix the problem at all. Okay?

So there's another example of when these sorts of diagnostics will help you figure out whether you should be fixing your optimization algorithm or fixing the optimization objective. Okay?

Let me think how much time I have. Hmm, let's see. Well okay, we have time. Let's do this. Show you one last example of a diagnostic. This is one that came up in, you know, my students' and my work on flying helicopters. This one actually, this example is the most complex of the three examples I'm gonna do today. I'm going to somewhat quickly, and this actually draws on reinforcement learning which is something that I'm not gonna talk about until towards – close to the end of the course here, but this just a more complicated example of a diagnostic we're gonna go over.

What I'll do is probably go over this fairly quickly, and then after we've talked about reinforcement learning in the class, I'll probably actually come back and redo this exact same example because you'll understand it more deeply. Okay?

So some of you know that my students and I fly autonomous helicopters, so how do you get a machine-learning algorithm to design the controller for helicopter? This is what we do. All right? This first step was you build a simulator for a helicopter, so, you know, there's a screenshot of our simulator. This is just like a – it's like a joystick simulator; you can fly a helicopter in simulation.

And then you choose a cost function, it's actually called a [inaudible] function, but for this actually I'll call it cost function. Say J of theta is, you know, the expected squared error in your helicopter's position. Okay? So this is J of theta is maybe it's expected square error or just the square error. And then we run a reinforcement-learning algorithm, you'll learn about RL algorithms in a few weeks. You run reinforcement learning algorithm in your simulator to try to minimize this cost function; try to minimize the squared error of how well you're controlling your helicopter's position. Okay?

The reinforcement learning algorithm will output some parameters, which I'm denoting theta subscript RL, and then you'll use that to fly your helicopter. So suppose you run this learning algorithm and you get out a set of controller parameters, theta subscript RL, that gives much worse performance than a human pilot. Then what do you do next? And in particular, you know, corresponding to the three steps above, there are three natural things you can try. Right?

You can try to – oh, the bottom of the slide got chopped off. You can try to improve the simulator. And maybe you think your simulator's isn't that accurate, you need to capture the aerodynamic effects more accurately. You need to capture the airflow and the turbulence affects around the helicopter more accurately. Maybe you need to modify the cost function. Maybe your square error isn't cutting it. Maybe what a human pilot does isn't just optimizing square area but it's something more subtle. Or maybe the reinforcement-learning algorithm isn't working; maybe it's not quite converging or something. Okay?

So these are the diagnostics that I actually used, and my students and I actually use to figure out what's going on. Actually, why don't you just think about this for a second and think what you'd do, and then I'll go on and tell you what we do. All right, so let me tell you what – how we do this and see whether it's the same as yours or not. And if you have a better idea than I do, let me know and I'll let you try it on my helicopter.

So here's a reasoning that I wanted to experiment, right. So, yeah, let's say the controller output by our reinforcement-learning algorithm does poorly. Well suppose the following three things hold true. Suppose the contrary, I guess. Suppose that the helicopter simulator is accurate, so let's assume we have an accurate

model of our helicopter. And let's suppose that the reinforcement learning algorithm, you know, correctly controls the helicopter in simulation, so we tend to run a learning algorithm in simulation so that, you know, the learning algorithm can crash a helicopter and it's fine. Right?

So let's assume our reinforcement-learning algorithm correctly controls the helicopter so as to minimize the cost function J of theta. And let's suppose that minimizing J of theta does indeed correspond to accurate or the correct autonomous flight. If all of these things held true, then that means that the parameters, theta RL, should actually fly well on my real helicopter. Right? And so the fact that the learning control parameters, theta RL, does not fly well on my helicopter, that sort of means that ones of these three assumptions must be wrong and I'd like to figure out which of these three assumptions is wrong. Okay?

So these are the diagnostics we use. First one is we look at the controller and see if it even flies well in simulation. Right? So the simulator of the helicopter that we did the learning on, and so if the learning algorithm flies well in the simulator but it doesn't fly well on my real helicopter, then that tells me the problem is probably in the simulator. Right? My simulator predicts the helicopter's controller will fly well but it doesn't actually fly well in real life, so could be the problem's in the simulator and we should spend out efforts improving the accuracy of our simulator.

Otherwise, let me write theta subscript human, be the human control policy. All right? So let's go ahead and ask a human to fly the helicopter, it could be in the simulator, it could be in real life, and let's measure, you know, the means squared error of the human pilot's flight. And let's see if the human pilot does better or worse than the learned controller, in terms of optimizing this objective function J of theta. Okay?

So if the human does worse, if even a very good human pilot attains a worse value on my optimization objective, on my cost function, than my learning algorithm, then the problem is in the reinforcement-learning algorithm. Because my reinforcement-learning algorithm was trying to minimize J of theta, but a human actually attains a lower value for J of theta than does my algorithm. And so that tells me that clearly my algorithm's not managing to minimize J of theta and that tells me the problem's in the reinforcement learning algorithm.

And finally, if J of theta – if the human actually attains a larger value for theta – excuse me, if the human actually attains a larger value for J of theta, the human actually has, you know, larger mean squared error for the helicopter position than does my reinforcement learning algorithms, that's I like – but I like the way the human flies much better than my reinforcement learning algorithm.

So if that holds true, then clearly the problem's in the cost function, right, because the human does worse on my cost function but flies much better than my learning algorithm. And so that means the problem's in the cost function. It means – oh excuse me, I meant minimizing it, not maximizing it, there's a typo on the slide, because that means that minimizing the cost function – my learning algorithm does a better job minimizing the cost function but doesn't fly as well as a human pilot. So that tells you that minimizing the cost function doesn't correspond to good autonomous flight. And what you should do it go back and see if you can change J of theta. Okay?

And so for those reinforcement learning problems, you know, if something doesn't work – often reinforcement learning algorithms just work but when they don't work, these are the sorts of diagnostics you use to figure out should we be focusing on the simulator, on changing the cost function, or on changing the reinforcement learning algorithm.

And again, if you don't know which of your three problems it is, it's entirely possible, you know, to spend two years, whatever, changing, building a better simulator for your helicopter. But it turns out that modeling helicopter aerodynamics is an active area of research. There are people, you know, writing entire PhD theses on this still. So it's entirely possible to go out and spend six years and write a PhD thesis and build a much better helicopter simulator, but if you're fixing the wrong problem it's not gonna help.

So quite often, you need to come up with your own diagnostics to figure out what's happening in an algorithm when something is going wrong. And unfortunately I don't know of – what I've described are sort of maybe some of the most common diagnostics that I've used, that I've seen, you know, to be useful for many problems. But very often, you need to come up with your own for your own specific learning problem.

And I just want to point out that even when the learning algorithm is working well, it's often a good idea to run diagnostics, like the ones I talked about, to make sure you really understand what's going on.

All right? And this is useful for a couple of reasons. One is that diagnostics like these will often help you to understand your application problem better. So some of you will, you know, graduate from Stanford and go on to get some amazingly high-paying job to apply machine-learning algorithms to some application problem of, you know, significant economic interest. Right?

And you're gonna be working on one specific important machine learning application for many months, or even for years. One of the most valuable things for you personally will be for you to get in − for you personally to get in an intuitive understanding of what works and what doesn't work your problem. Sort of right now in the industry, in Silicon Valley or around the world, there are many companies with important machine learning problems and there are often people working on the same machine learning problem, you know, for many months or for years on end.

And when you're doing that, I mean solving a really important problem using learning algorithms, one of the most valuable things is just your own personal intuitive understanding of the problem. Okay? And diagnostics, like the sort I talked about, will be one way for you to get a better and better understanding of these problems.

It turns out, by the way, there are some of Silicon Valley companies that outsource their machine learning. So there's sometimes, you know, whatever. They're a company in Silicon Valley and they'll, you know, hire a firm in New York to run all their learning algorithms for them.

And I'm not a businessman, but I personally think that's often a terrible idea because if your expertise, if your understanding of your data is given, you know, to an outsource agency, then if you don't maintain that expertise, if there's a problem you really care about then it'll be your own, you know, understanding of the problem that you build up over months that'll be really valuable. And if that knowledge is outsourced, you don't get to keep that knowledge yourself. I personally think that's a terrible idea. But I'm not a businessman, but I just see people do that a lot, and just.

Let's see. Another reason for running diagnostics like these is actually in writing research papers, right? So diagnostics and error analyses, which I'll talk about in a minute, often help to convey insight about the problem and help justify your research claims.

So for example, rather than writing a research paper, say, that's says, you know, "Oh well here's an algorithm that works. I built this helicopter and it flies," or whatever, it's often much more interesting to say, "Here's an algorithm that works, and it works because of a specific component X. And moreover, here's the diagnostic that gives you justification that shows X was the thing that fixed this problem," and that's where you made it work. Okay?

So that leads me into a discussion on error analysis, which is often good machine learning practice, is a way for understanding what your sources of errors are. So what I call error analyses − and let's check questions about this. Yeah?

Student: What ended up being wrong with the helicopter?

Instructor (Andrew Ng):Oh, don't know. Let's see. We've flown so many times. The thing that is most difficult a helicopter is actually building a very − I don't know. It changes all the time. Quite often, it's actually the simulator. Building an accurate simulator of a helicopter is very hard. Yeah. Okay.

So for error analyses, this is a way for figuring out what is working in your algorithm and what isn't working. And we're gonna talk about two specific examples. So there are many learning − there are many sort of IA systems, many machine learning systems, that combine many different components into a pipeline. So here's sort of a contrived example for this, not dissimilar in many ways from the actual machine learning systems you see.

So let's say you want to recognize people from images. This is a picture of one of my friends. So you take this input in camera image, say, and you often run it through a long pipeline. So for example, the first thing you may do may be preprocess the image and remove the background, so you remove the background. And then you run a face detection algorithm, so a machine learning algorithm to detect people's faces. Right?

And then, you know, let's say you want to recognize the identity of the person, right, this is your application. You then segment of the eyes, segment of the nose, and have different learning algorithms to detect the mouth and so on. I know; she might not want to be friend after she sees this. And then having found all these features, based on, you know, what the nose looks like, what the eyes looks like, whatever,

then you feed all the features into a logistic regression algorithm. And your logistic regression or soft match regression, or whatever, will tell you the identity of this person. Okay?

So this is what error analysis is. You have a long complicated pipeline combining many machine learning components. Many of these would be used in learning algorithms. And so, it's often very useful to figure out how much of your error can be attributed to each of these components. So what we'll do in a typical error analysis procedure is we'll repeatedly plug in the ground-truth for each component and see how the accuracy changes.

So what I mean by that is the figure on the bottom left – bottom right, let's say the overall accuracy of the system is 85 percent. Right? Then I want to know where my 15 percent of error comes from. And so what I'll do is I'll go to my test set and I'll actually code it and – oh, instead of – actually implement my correct background removal. So actually, go in and give it, give my algorithm what is the correct background versus foreground.

And if I do that, let's color that blue to denote that I'm giving that ground-truth data in the test set, let's assume our accuracy increases to 85.1 percent. Okay? And now I'll go in and, you know, give my algorithm the ground-truth, face detection output. So I'll go in and actually on my test set I'll just tell the algorithm where the face is. And if I do that, let's say my algorithm's accuracy increases to 91 percent, and so on.

And then I'll go for each of these components and just give it the ground-truth label for each of the components, because say, like, the nose segmentation algorithm's trying to figure out where the nose is. I just in and tell it where the nose is so that it doesn't have to figure that out. And as I do this, one component through the other, you know, I end up giving it the correct output label and end up with 100 percent accuracy.

And now you can look at this table – I'm sorry this is cut off on the bottom, it says logistic regression 100 percent. Now you can look at this table and see, you know, how much giving the ground-truth labels for each of these components could help boost your final performance. In particular, if you look at this table, you notice that when I added the face detection ground-truth, my performance jumped from 85.1 percent accuracy to 91 percent accuracy. Right?

So this tells me that if only I can get better face detection, maybe I can boost my accuracy by 6 percent. Whereas in contrast, when I, you know, say plugged in better, I don't know, background removal, my accuracy improved from 85 to 85.1 percent. And so, this sort of diagnostic also tells you that if your goal is to improve the system, it's probably a waste of your time to try to improve your background subtraction. Because if even if you got the ground-truth, this is gives you, at most, 0.1 percent accuracy, whereas if you do better face detection, maybe there's a much larger potential for gains there. Okay?

So this sort of diagnostic, again, is very useful because if your is to improve the system, there are so many different pieces you can easily choose to spend the next three months on. Right? And choosing the right piece is critical, and this sort of diagnostic tells you what's the piece that may actually be worth your time to work on.

There's sort of another type of analyses that's sort of the opposite of what I just talked about. The error analysis I just talked about tries to explain the difference between the current performance and perfect performance, whereas this sort of ablative analysis tries to explain the difference between some baselines, some really bad performance and your current performance.

So for this example, let's suppose you've built a very good anti-spam classifier for adding lots of clever features to your logistic regression algorithm. Right? So you added features for spam correction, for, you know, sender host features, for email header features, email text parser features, JavaScript parser features, features for embedded images, and so on.

So now let's say you preview the system and you want to figure out, you know, how well did each of these – how much did each of these components actually contribute? Maybe you want to write a research paper and claim this was the piece that made the big difference. Can you actually document that claim and justify it?

So in ablative analysis, here's what we do. So in this example, let's say that simple logistic regression without any of your clever improvements get 94 percent performance. And you want to figure out what accounts for your improvement from 94 to 99.9 percent performance. So in ablative analysis and so instead

of adding components one at a day, we'll instead remove components one at a time to see how it rates.

So start with your overall system, which is 99 percent accuracy. And then we remove spelling correction and see how much performance drops. Then we'll remove the sender host features and see how much performance drops, and so on. All right? And so, in this contrived example, you see that, I guess, the biggest drop occurred when you remove the text parser features. And so you can then make a credible case that, you know, the text parser features where what really made the biggest difference here. Okay?

And you can also tell, for instance, that, I don't know, removing the sender host features on this line, right, performance dropped from 99.9 to 98.9. And so this also means that in case you want to get rid of the sender host features to speed up computational something that would be a good candidate for elimination. Okay?

Student: Are there any guarantees that if you shuffle around the order in which you drop those features that you'll get the same –

Instructor (Andrew Ng): Yeah. Let's address the question: What if you shuffle in which you remove things? The answer is no. There's no guarantee you'd get the similar result. So in practice, sometimes there's a fairly natural of ordering for both types of analyses, the error analyses and ablative analysis, sometimes there's a fairly natural ordering in which you add things or remove things, sometimes there's isn't. And quite often, you either choose one ordering and just go for it or –

And don't think of these analyses as sort of formulas that are constants, though; I mean feel free to invent your own, as well. You know one of the things that's done quite often is take the overall system and just remove one and then put it back, then remove a different one then put it back until all of these things are done. Okay.

So the very last thing I want to talk about is sort of this general advice for how to get started on a learning problem. So here's a cartoon description on two broad to get started on learning problem. The first one is carefully design your system, so you spend a long time designing exactly the right features, collecting the right data set, and designing the right algorithmic structure, then you implement it and hope it works. All right?

The benefit of this sort of approach is you get maybe nicer, maybe more scalable algorithms, and maybe you come up with new elegant learning algorithms. And if your goal is to, you know, contribute to basic research in machine learning, if your goal is to invent new machine learning algorithms, this process of slowing down and thinking deeply about the problem, you know, that is sort of the right way to go about is think deeply about a problem and invent new solutions.

Second sort of approach is what I call build-and-fix, which is we input something quick and dirty and then you run error analyses and diagnostics to figure out what's wrong and you fix those errors.

The benefit of this second type of approach is that it'll often get your application working much more quickly. And especially with those of you, if you end up working in a company, and sometimes – if you end up working in a company, you know, very often it's not the best product that wins; it's the first product to market that wins. And so there's – especially in the industry. There's really something to be said for, you know, building a system quickly and getting it deployed quickly.

And the second approach of building a quick-and-dirty, I'm gonna say hack and then fixing the problems will actually get you to a system that works well much more quickly. And the reason is very often it's really not clear what parts of a system are easier to think of to build and therefore what you need to spends lot of time focusing on.

So there's that example I talked about just now. Right? For identifying people, say. And with a big complicated learning system like this, a big complicated pipeline like this, it's really not obvious at the outset which of these components you should spend lots of time working on. Right? And if you didn't know that preprocessing wasn't the right component, you could easily have spent three months working on better background subtraction, not knowing that it's just not gonna ultimately matter.

And so the only way to find out what really works was inputting something quickly and you find out what parts – and find out what parts are really the hard parts to implement, or what parts are hard parts that could make a difference in performance. In fact, say that if your goal is to build a people recognition system, a system like this is actually far too complicated as your initial system. Maybe after you're prototyped a

few systems, and you converged a system like this. But if this is your first system you're designing, this is much too complicated.

Also, this is a very concrete piece of advice, and this applies to your projects as well. If your goal is to build a working application, Step 1 is actually probably not to design a system like this. Step 1 is where you would plot your data. And very often, and if you just take the data you're trying to predict and just plot your data, plot X, plot Y, plot your data everywhere you can think of, you know, half the time you look at it and go, "Gee, how come all those numbers are negative? I thought they should be positive. Something's wrong with this dataset."

And it's about half the time you find something obviously wrong with your data or something very surprising. And this is something you find out just by plotting your data, and that you won't find out be implementing these big complicated learning algorithms on it. Plotting the data sounds so simple, it was one of the pieces of advice that lots of us give but hardly anyone follows, so you can take that for what it's worth.

Let me just reiterate, what I just said here may be bad advice if your goal is to come up with new machine learning algorithms. All right? So for me personally, the learning algorithm I use the most often is probably logistic regression because I have code lying around. So give me a learning problem, I probably won't try anything more complicated than logistic regression on it first. And it's only after trying something really simple and figure our what's easy, what's hard, then you know where to focus your efforts.

But again, if your goal is to invent new machine learning algorithms, then you sort of don't want to hack up something and then add another hack to fix it, and hack it even more to fix it. Right? So if your goal is to do novel machine learning research, then it pays to think more deeply about the problem and not gonna follow this specifically.

Shoot, you know what? All right, sorry if I'm late but I just have two more slides so I'm gonna go through these quickly. And so, this is what I think of as premature statistical optimization, where quite often, just like premature optimization of code, quite often people will prematurely optimize one component of a big complicated machine learning system. Okay?

Just two more slides. This was – this is a sort of cartoon that highly influenced my own thinking. It was based on a paper written by Christos Papadimitriou. This is how progress – this is how developmental progress of research often happens. Right? Let's say you want to build a mail delivery robot, so I've drawn a circle there that says mail delivery robot. And it seems like a useful thing to have. Right? You know free up people, don't have to deliver mail.

So what – to deliver mail, obviously you need a robot to wander around indoor environments and you need a robot to manipulate objects and pickup envelopes. And so, you need to build those two components in order to get a mail delivery robot. And so I've drawing those two components and little arrows to denote that, you know, obstacle avoidance is needed or would help build your mail delivery robot.

Well for obstacle for avoidance, clearly, you need a robot that can navigate and you need to detect objects so you can avoid the obstacles. Now we're gonna use computer vision to detect the objects. And so, we know that, you know, lighting sometimes changes, right, depending on whether it's the morning or noontime or evening. This is lighting causes the color of things to change, and so you need an object detection system that's invariant to the specific colors of an object. Right? Because lighting changes, say.

Well color, or RGB values, is represented by three-dimensional vectors. And so you need to learn when two colors might be the same thing, when two, you know, visual appearance of two colors may be the same thing as just the lighting change or something. And to understand that properly, we can go out and study differential geometry of 3d manifolds because that helps us build a sound theory on which to develop our 3d similarity learning algorithms.

And to really understand the fundamental aspects of this problem, we have to study the complexity of non-Riemannian geometries. And on and on it goes until eventually you're proving convergence bounds for sampled of non-monotonic logic. I don't even know what this is because I just made it up. Whereas in reality, you know, chances are that link isn't real. Color variance just barely helped object recognition maybe. I'm making this up. Maybe differential geometry was hardly gonna help 3d similarity learning and that link's also gonna fail. Okay?

So, each of these circles can represent a person, or a research community, or a thought in your head. And there's a very real chance that maybe there are all these papers written on differential geometry of 3d manifolds, and they are written because some guy once told someone else that it'll help 3d similarity learning.

And, you know, it's like "A friend of mine told me that color invariance would help in object recognition, so I'm working on color invariance. And now I'm gonna tell a friend of mine that his thing will help my problem. And he'll tell a friend of his that his thing will help with his problem." And pretty soon, you're working on convergence bound for sampled non-monotonic logic, when in reality none of these will see the light of day of your mail delivery robot. Okay?

I'm not criticizing the role of theory. There are very powerful theories, like the theory of VC dimension, which is far, far, far to the right of this. So VC dimension is about as theoretical as it can get. And it's clearly had a huge impact on many applications. And there's, you know, dramatically advanced data machine learning. And another example is theory of NP-hardness as again, you know, is about theoretical as it can get. It's like a huge application on all of computer science, the theory of NP-hardness.

But when you are off working on highly theoretical things, I guess, to me personally it's important to keep in mind are you working on something like VC dimension, which is high impact, or are you working on something like convergence bound for sampled non-monotonic logic, which you're only hoping has some peripheral relevance to some application. Okay?

For me personally, I tend to work on an application only if I – excuse me. For me personally, and this is a personal choice, I tend to trust something only if I personally can see a link from the theory I'm working on all the way back to an application. And if I don't personally see a direct link from what I'm doing to an application then, you know, then that's fine. Then I can choose to work on theory, but I wouldn't necessarily trust that what the theory I'm working on will relate to an application, if I don't personally see a link all the way back.

Just to summarize. One lesson to take away from today is I think time spent coming up with diagnostics for learning algorithms is often time well spent. It's often up to your own ingenuity to come up with great diagnostics. And just when I personally, when I work on machine learning algorithm, it's not uncommon for me to be spending like between a third and often half of my time just writing diagnostics and trying to figure out what's going right and what's going on.

Sometimes it's tempting not to, right, because you want to be implementing learning algorithms and making progress. You don't want to be spending all this time, you know, implementing tests on your learning algorithms; it doesn't feel like when you're doing anything. But when I implement learning algorithms, at least a third, and quite often half of my time, is actually spent implementing those tests and you can figure out what to work on. And I think it's actually one of the best uses of your time.

Talked about error analyses and ablative analyses, and lastly talked about, you know, different approaches and the risks of premature statistical optimization. Okay.

Sorry I ran you over. I'll be here for a few more minutes for your questions. That's [inaudible] today.

[End of Audio]

Duration: 82 minutes

1.12 Machine Learning Lecture 12[12]

Figure 1.12

MachineLearning-Lecture12

Instructor (Andrew Ng): Okay. Good morning. I just have one quick announcement of sorts. So many of you know that it was about two years ago that Stanford submitted an entry to the DARPA Grand Challenge which was the competition to build a car to drive itself across the desert. So some of you may know that this weekend will be the next DARPA Grand Challenge phase, and so Stanford – the team that – one of my colleagues Sebastian Thrun has a team down in OA now and so they'll be racing another autonomous car.

So this is a car that incorporates many tools and AI machines and everything and so on, and it'll try to drive itself in midst of traffic and avoid other cars and carry out the sort of mission. So if you're free this weekend – if you're free on Saturday, watch TV or search online for Urban Challenge, which is the name of the competition. It should be a fun thing to watch, and it'll hopefully be a cool demo or instance of AI and machines in action.

Let's see. My laptop died a few seconds before class started so let me see if I can get that going again. If not, I'll show you the things I have on the blackboard instead. Okay. So good morning and welcome back. What I want to do today is actually begin a new chapter in 229 in which I'm gonna start to talk about [inaudible]. So [inaudible] today is I'm gonna just very briefly talk about clustering's, [inaudible] algorithm. [Inaudible] and a special case of the EM, Expectation Maximization, algorithm with a mixture of [inaudible] model to describe something called Jensen and Equality and then we'll use that derive a general form of something called the EM or the Expectation Maximization algorithm, which is a very useful algorithm. We sort of use it all over the place and different unsupervised machine or any application. So the cartoons that I used to draw for supervised learning was you'd be given the data set like this, right, and you'd use [inaudible] between the positive and negative crosses and we'd call it the supervised learning because you're sort of told what the right cross label is for every training example, and that was the supervision. In unsupervised learning, we'll study a different problem. You're given a data set that maybe just comprises a set of points. You're just given a data set with no labels and no indication of what the "right answers" or where the supervision is and it's the job of the algorithm to discover structure in the data.

So in this lecture and the next couple of weeks we'll talk about a variety of unsupervised learning algorithms that can look at data sets like these and discover there's different types of structure in it. In this particular cartoon that I've drawn – one has the structure that you and I can probably see as is that this data lives in two different crosses and so the first unsupervised learning algorithm that I'm just gonna talk about will be a clustering algorithm. It'll be an algorithm that looks for a data set like this and automatically breaks the data set into different smaller clusters. So let's see. When my laptop comes back up, I'll show you an example. So clustering algorithms like these have a variety of applications. Just to rattle off a few of the better-known ones I guess in biology application you often cross the different things here. You have [inaudible] genes and they cluster the different genes together in order to examine them and understand the biological function better. Another common application of clustering is market research. So imagine you have a customer database of how your different customers behave. It's a very common practice to apply clustering algorithms to break your database of customers into different market segments so that you can target your products towards different market segments and target your sales pitches specifically to different market segments.

Something we'll do later today – I don't want to do this now, but you actually go to a website, like, use.google.com and that's an example of a website that uses a clustering algorithm to everyday group related news articles together to display to you so that you can see one of the thousand news articles today on whatever the top story of today is and all the 500 news articles on all the different websites on different story of the day. And a very solid [inaudible] actually talks about image segmentation, which the application of when you might take a picture and group together different subsets of the picture into coherent pieces of pixels to try to understand what's contained in the picture. So that's yet another application of clustering. The next idea is given a data set like this, given a set of points, can you automatically group the data sets into coherent clusters. Let's see. I'm still waiting for the laptop to come back so I can show you an example. You know what, why don't I just start to write out the specific clustering algorithm and then I'll show you the animation later. So this is the called the k-means clustering algorithm for finding clustering's near the

inset. The input to the algorithm will be an unlabeled data set which I write as X1, X2, [inaudible] and because we're now talking about unsupervised learning, you see a lot of this as [inaudible] with just the Xs and no cross labels Y. So what a k-means algorithm does is the following.

This will all make a bit more sense when I show you the animation on my laptop. To initialize a set of points, called the cluster centroids, [inaudible] randomly and so if you're [inaudible] of training data are [inaudible] then your cluster centroids, these muse, will also be vectors and [inaudible] and then you repeat until convergence the following two steps. So the cluster centroids will be your guesses for where the centers of each of the clusters are and so in one of those steps you look at each point, XI and you look at which cluster centroid J is closest to it and then this step is called assigning your point XI to cluster J. So looking at each point and picking the cluster centroid that's closest to it and the other step is you update the cluster centroids to be the median of all the points that have been assigned to it. Okay. Let's see. Could you please bring down the display for the laptop? Excuse me. Okay. Okay. There we go. Okay. So here's an example of the k-means algorithm and hope the animation will make more sense. This is an inch chopped off. This is basically an example I got from Michael Jordan in Berkley. So these points in green are my data points and so I'm going to randomly initialize a pair of cluster centroids. So the [inaudible] blue crosses to note the positions of New1 and New2 say if I'm going to guess that there's two clusters in this data. Sets of k-means algorithms as follow. I'm going to repeatedly go to all of the points in my data set and I'm going to associate each of the green dots with the closer of the two cluster centroids, so visually, I'm gonna denote that by painting each of the dots either blue or red depending on which is the closer cluster centroid. Okay.

So all the points closer to the blue cross points are painted blue and so on. The other step is updating the cluster centroids and so I'm going to repeatedly look at all the points that I've painted blue and compute the average of all of the blue dots, and I'll look at all the red dots and compute the average of all the red dots and then I move the cluster centroids as follows to the average of the respective locations. So this is now [inaudible] of k-means on here, and now I'll repeat the same process. I look at all the points and assign all the points closer to the blue cross to the color blue and similarly red. And so now I have that assignments of points to the cluster centroids and finally, I'll again compute the average of all the blue points and compute the average of all the red points and update the cluster centroids again as follows and now k-means is actually [inaudible]. If you keep running these two sets of k-means over and over, the cluster centroids and the assignment of the points closest to the cluster centroids will actually remain the same. Yeah.

Student:[Inaudible]

Instructor (Andrew Ng):Yeah, I'll assign that in a second. Yeah. Okay. So [inaudible]. Take a second to look at this again and make sure you understand how the algorithm I wrote out maps onto the animation that we just saw. Do you have a question?

Student:[Inaudible]

Instructor (Andrew Ng):I see. Okay. Let me answer on that in a second. Okay. So these are the two steps. This step 2.1 was assigning the points to the closest centroid and 2.2 was shifting the cluster centroid to be the mean of all the points assigned to that cluster centroid. Okay. Okay. [Inaudible] questions that we just had, one is, does the algorithm converge? The answer is yes, k-means is guaranteed to converge in a certain sense. In particular, if you define the distortion function to be J of C [inaudible] squared. You can define the distortion function to be a function of the cluster assignments, and the cluster centroids and [inaudible] square distances, which mean the points and the cluster centroids that they're assigned to, then you can show – I won't really prove this here but you can show that k-means is called [inaudible] on the function J. In particular, who remembers, it's called in a sense as an authorization algorithm, I don't know, maybe about two weeks ago, so called in a sense is the algorithm that we'll repeatedly [inaudible] with respect to C. Okay. So that's called [inaudible]. And so what you can prove is that k-means – the two steps of k-means, are exactly optimizing this function with respect to C and will respect a new alternately. And therefore, this function, J of C, new, must be decreasing monotonically on every other variation and so the sense in which k-means converges is that this function, J of C, new, can only go down and therefore, this function will actually eventually converge in the sense that it will stop going down.

Okay. It's actually possible that there may be several clustering's they give the same value of J of C, new

and so k-means may actually switch back and forth between different clustering's that they [inaudible] in the extremely unlikely case, if there's multiple clustering's, they give exactly the same value for this objective function. K-means may also be [inaudible] it'll just never happen. That even if that happens, this function J of C, new will converge. Another question was how do you choose the number of clusters? So it turns out that in the vast majority of time when people apply k-means, you still just randomly pick a number of clusters or you randomly try a few different numbers of clusters and pick the one that seems to work best. The number of clusters in this algorithm instead of just one parameters, so usually I think it's not very hard to choose automatically. There are some automatic ways of choosing the number of clusters, but I'm not gonna talk about them. When I do this, I usually just pick of the number of clusters randomly. And the reason is I think for many clustering problems the "true" number of clusters is actually ambiguous so for example if you have a data set that looks like this, some of you may see four clusters, right, and some of you may see two clusters, and so the right answer for the actual number of clusters is sort of ambiguous. Yeah.

Student:[Inaudible]. [Inaudible] clusters [inaudible] far away from the data point [inaudible] points and the same cluster?

Instructor (Andrew Ng):I see. Right. So yes. K-means is susceptible to [inaudible] so this function, J of C, new is not a convex function and so k-means, sort of called in a sense on the non-convex function is not guaranteed to converge the [inaudible]. So k-means is susceptible to local optimal and [inaudible]. One thing you can do is try multiple random initializations and then run clustering a bunch of times and pick the solution that ended up with the lowest value for the distortion function. Yeah.

Student:[Inaudible]

Instructor (Andrew Ng): Yeah, let's see. Right. So what if one cluster centroid has no points assigned to it, again, one thing you could do is just eliminate it exactly the same. Another thing you can is you can just reinitialize randomly if you really [inaudible]. More questions. Yeah.

Student:[Inaudible] as a norm or can you [inaudible] or infinity norm or −

Instructor (Andrew Ng):I see. Right. Is it usually two norms? Let's see. For the vast majority of applications I've seen for k-means, you do take two norms when you have data [inaudible]. I'm sure there are others who have taken infinity norm and one norm as well. I personally haven't seen that very often, but there are other variations on this algorithm that use different norms, but the one I described is probably the most commonly used there is. Okay. So that was k-means clustering. What I want to do next and this will take longer to describe is actually talk about a closely related problem. In particular, what I wanted to do was talk about density estimation. As another k-means example, this is a problem that I know some guys that worked on. Let's say you have aircraft engine building off an assembly. Let's say you work for an aircraft company, you're building aircraft engines off the assembly line and as the aircraft engines roll off the assembly line, you test these aircraft engines and measure various different properties of it and to use [inaudible] example I'm gonna write these properties as heat and vibrations. Right.

In reality, you'd measure different vibrations, different frequencies and so on. We'll just write the amount of heat produced and vibrations produced. Let's say that maybe it looks like that and what you would like to do is estimate the density of these [inaudible] of the joint distribution, the amount of heat produced and the amount of vibrations because you would like to detect [inaudible] so that as a new aircraft engine rolls off the assembly line, you can then measure the same heat and vibration properties. If you get a point there, you can then ask, "How likely is it that there was an undetected flaw in this aircraft engine that it needs to go undergo further inspections?" And so if we look at the typical distribution of features we get, and we build a model for P of X and then if P of X is very small for some new aircraft engine then that would raise a red flag and we'll say there's an anomaly aircraft engine and we should subject it to further inspections before we let someone fly with the engine. So this problem I just described is an instance of what is called anomaly detection and so a common way of doing anomaly detection is to take your training set and from this data set, build a model, P of X of the density of the typical data you're saying and if you ever then see an example with very low probability under P of X, then you may flag that as an anomaly example.

Okay? So anomaly detection is also used in security applications. If many, very unusual transactions to start to appear on my credit card, that's a sign to me that someone has stolen my credit card. And what I want to do now is talk about specific algorithm for density estimation, and in particular, one that works with

data sets like these, that, you know, this distribution like this doesn't really fall into any of the standard text book distributions. This is not really, like, a Gaussian or a [inaudible] explanation or anything. So can we come up with a model to estimate densities that may look like these somewhat unusual shapes? Okay. So to describe the algorithm a bit a more I'm also going to use a one dimensional example rather than a two D example, and in the example that I'm going to describe I'm going to say that let's imagine maybe a data set that looks like this where the horizontal access here is the X axis and these dots represent the positions of the data set that I have. Okay. So this data set looks like it's maybe coming from a density that looks like that as if this was the sum of two Gaussian distributions and so the specific model I'm gonna describe will be what's called a mixture of Gaussian's model.

And just be clear that the picture I have is that when visioning that maybe there were two separate Gaussian's that generated this data set, and if only I knew what the two Gaussian's were, then I could put a Gaussian to my crosses, put a Gaussian to the Os and then sum those up to get the overall density for the two, but the problem is I don't actually have access to these labels. I don't actually know which of the two Gaussian's each of my data points came from and so what I'd like to do is come up with an algorithm to fit this mixture of Gaussian's model even when I don't know which of the two Gaussian's each of my data points came from. Okay. So here's the idea. In this model, I'm going to imagine there's a latent random variable, latent is just synonymous with hidden or unobserved, okay. So we're gonna imagine there's a latent random variable Z and XI, ZI have a joint distribution that is given as follows. We have that P of X, ZI by the chamber of probability, this is always like that. This is always true. And moreover, our [inaudible] is given by the following ZI is distributed multinomial with parameters I. And in the special case where I have just to make sure that two Gaussian's and ZI will be [inaudible], and so these parameter [inaudible] are the parameters of a multinomial distribution.

And the distribution of XI conditioned on ZI being equal to J so it's P of XI given ZI is equal to J. That's going to be a Gaussian distribution with [inaudible] and covariant sigler. Okay. So this should actually look extremely familiar to you. What I've written down are pretty much the same equations that I wrote down for the Gaussian Discriminant Analysis algorithm that we saw way back, right, except that the differences – instead of, I guess supervised learning where we were given the cross labels Y, I've now replaced Y in Gaussian Discriminant Analysis with these latent random variables or these unobserved random variables Z, and we don't actually know what the values of Z are. Okay. So just to make the link to the Gaussian Discriminant Analysis even a little more explicit – if we knew what the Zs were, which was actually don't, but suppose for the sake of argument that we actually knew which of, say the two Gaussian's, each of our data points came from, then you can use [inaudible] estimation – you can write down the likelihood the parameters which would be that and you can then use [inaudible] estimation and you get exactly the same formula as in Gaussian Discriminant Analysis. Okay. So if you knew the value of the Z, you can write down the law of likelihood and do maximum likeliness this way, and you can then estimate all the parameters of your model. Does this make sense? Raise your hand if this makes sense. Cool. Some of you have questions? Some of you didn't raise your hands. Yeah.

Student: So this ZI is just a label, like, an X or an O?

Instructor (Andrew Ng):Yes. Basically. Any other questions? Okay. So if you knew the values of Z, the Z playing a similar role to the cross labels in Gaussian's Discriminant Analysis, then you could use maximum likeliness estimation parameters. But in reality, we don't actually know the values of the Zs. All we're given is this unlabeled data set and so let me write down the specific bootstrap procedure in which the idea is that we're going to use our model to try and guess what the values of Z is. We don't know our Z, but we'll just take a guess at the values of Z and we'll then use some of the values of Z that we guessed to fit the parameters of the rest of the model and then we'll actually iterate. And now that we have a better estimate for the parameters for the rest of the model, we'll then take another guess for what the values of Z are. And then we'll sort of use something like the maximum likeliness estimation to set even parameters of the model. So the algorithm I'm gonna write down is called the EM Algorithm and it proceeds as follows. Repeat until convergence and the E set, we're going to guess the values of the unknown ZIs and in particular, I'm going to set WIJ. Okay. So I'm going to compute the probability that ZI is equal to J. So I'm going to use the rest of the parameters in my model and then I'm gonna compute the probability that point XI came from

Gaussian number J. And just to be sort of concrete about what I mean by this, this means that I'm going to compute P of XI.

This step is sort of [inaudible], I guess. And again, just to be completely concrete about what I mean about this, the [inaudible] rate of P of XI given ZI equals J, you know, well that's the Gaussian density. Right? That's one over E to the – [inaudible] and then divided by sum from O equals 1 to K of [inaudible] of essentially the same thing, but with J replaced by L. Okay. [Inaudible] for the Gaussian and the numerator and the sum of the similar terms of the denominator. Excuse me. This is the sum from O equals 1 through K in the denominator. Okay. Let's see. The maximization step where you would then update your estimates of the parameters. So I'll just lay down the formulas here. When you see these, you should compare them to the formulas we had for maximum likelihood estimation. And so these two formulas on top are very similar to what you saw for Gaussian Discriminant Analysis except that now, we have these [inaudible] so WIJ is – you remember was the probability that we computed that point I came from Gaussian's. I don't want to call it cluster J, but that's what – point I came from Gaussian J, rather than an indicator for where the point I came from Gaussian J. Okay. And the one slight difference between this and the formulas who have a Gaussian's Discriminant Analysis is that in the mixture of Gaussian's, we more commonly use different covariant [inaudible] for the different Gaussian's.

So in Gaussian's Discriminant Analysis, sort of by convention, you usually model all of the crosses to the same covariant matrix sigma. I just wrote down a lot of equations. Why don't you just take a second to look at this and make sure it all makes sense? Do you have questions about this? Raise your hand if this makes sense to you? [Inaudible]. Okay. Only some of you. Let's see. So let me try to explain that a little bit more. Some of you recall that in Gaussian's Discriminant Analysis, right, if we knew the values for the ZIs so let's see. Suppose I was to give you labeled data sets, suppose I was to tell you the values of the ZIs for each example, then I'd be giving you a data set that looks like this. Okay. So here's my 1 D data set. That's sort of a typical 1 D Gaussian's Discriminant Analysis. So for Gaussian's Discriminant Analysis we figured out the maximum likeliness estimation and the maximum likeliness estimate for the parameters of GDA, and one of the estimates for the parameters for GDA was [inaudible] which is the probability that ZI equals J. You would estimate that as sum of I equals sum of I from 1 to M indicator ZI equals J and divide by N. Okay. When we're deriving GDA, [inaudible]. If you knew the cross labels for every example you cross, then this was your maximum likeliness estimate for the chance that the labels came from the positive [inaudible] versus the negative [inaudible]. It's just a fraction of examples.

Your maximum likeliness estimate for probability of getting examples from cross J is just the fraction of examples in your training set that actually came from cross J. So this is the maximum likeliness estimation for Gaussian's Discriminant Analysis. Now, in the mixture of Gaussian's model and the EM problem we don't actually have these cross labels, right, we just have an unlabeled data set like this. We just have a set of dots. I'm trying to draw the same data set that I had above, but just with the cross labels. So now, it's as if you only get to observe the XIs, but the ZIs are unknown. Okay. So the cross label is unknown. So in the EM algorithm we're going to try to take a guess for the values of the ZIs, and specifically, in the E step we computed WIJ was our current best guess for the probability that ZI equals J given that data point. Okay. So this just means given my current hypothesis, the way the Gaussian's are, and given everything else, can I compute the [inaudible] probability – what was the [inaudible] probability that the point XI actually came from cross J? What is the probability that this point was a cross versus O? What's the probability that this point was [inaudible]? And now in the M step, my formula of estimating for the parameters [inaudible] will be given by 1 over M sum from I equals 1 through M, sum of WIJ. So WIJ is right. The probability is my best guess for the probability that point I belongs to Gaussian or belongs to cross J, and [inaudible] using this formula instead of this one. Okay.

And similarly, this is my formula for the estimate for new J and it replaces the WIJs with these new indicator functions, you get back to the formula that you had in Gaussian's Discriminant Analysis. I'm trying to convey an intuitive sense of why these algorithm's make sense. Can you raise your hand if this makes sense now? Cool. Okay. So what I want to do now is actually present a broader view of the EM algorithm. What you just saw was a special case of the EM algorithm for specially to make sure of Gaussian's model, and in the remaining half hour I have today I'm going to describe a general description of the EM

algorithm and everything you just saw will be devised, sort of there's a special case of this more general view that I'll present now. And as a pre-cursor to actually deriving this more general view of the EM algorithm, I'm gonna have to describe something called Jensen's and Equality that we use in the derivation.

So here's Jensen's and Equality. Just let F be a convex function. So a function is a convex of the second derivative, which I've written F prime prime to [inaudible]. The functions don't have to be differentiable to be convex, but if it has a second derivative, then F prime prime should be creating a 0. And let X be a random variable then the F applied to the expectation of X is less than the equal of 2D expectation of F of F. Okay. And hopefully you remember I often drop the square back, so E of X is the [inaudible], I'll often drop the square brackets.

So let me draw a picture that would explain this and I think – Many of my friends and I often don't remember is less than or great than or whatever, and the way that many of us remember the sign of that in equality is by drawing the following picture. For this example, let's say, X is equal to 1 with a probability of one-half and X is equal to 6 worth probability 1 whole. So I'll illustrate this inequality with an example. So let's see. So X is 1 with probability one-half and X is 6 with probably with half and so the expected value of X is 3.5. It would be in the middle here. So that's the expected value of X. The horizontal axis here is the X axis. And so F of the expected value of X, you can read of as this point here. So this is F of the expected value of X. Where as in contrast, let's see. If X is equal to 1 then here's F of 1 and if X equaled a 6 then here's F of 6 and the expected value of F of X, it turns out, is now averaging on the vertical axis. We're 50 percent chance you get F of 1 with 50 percent chance you get F of 6 and so these expected value of F of X is the average of F of 1 and F of 6, which is going to be the value in the middle here. And so in this example you see that the expected value of F of X is greater than or equal to F of the expected value of X. Okay.

And it turns out further that if F double prime of X makes [inaudible] than Z row, if this happens, we say F is strictly convex then the inequality holds an equality or in other words, E of F of X equals F of EX, if and only if, X is a constant with probability 1. Well, another way of writing this is X equals EX. Okay. So in other words, if F is a strictly convex function, then the only way for this inequality to hold its equality is if the random variable X always takes on the same value. Okay. Any questions about this? Yeah.

Student:

[Inaudible]

Instructor (Andrew Ng): Say that again?

Student: What is the strict [inaudible]?

Instructor (Andrew Ng):I still couldn't hear that. What is –

Student: What is the strictly convex [inaudible]?

Instructor (Andrew Ng): Oh, I see. If double prime of X is strictly greater than 0 that's my definition for strictly convex. If the second derivative of X is strictly greater than 0 then that's what it means for F to be strictly convex.

Student:

[Inaudible]

Instructor (Andrew Ng):I see. Sure. So for example, this is an example of a convex function that's not strictly convexed because there's part of this function is a straight line and so F double prime would be zero in this portion. Let's see. Yeah. It's just a less formal way of saying strictly convexed just means that you can't have a convex function within a straight line portion and then [inaudible]. Speaking very informally, think of this as meaning that there aren't any straight line portions. Okay. So here's the derivation for the general version of EM. The problem was face is as follows. We have some model for the joint distribution of X of Z, but we observe only X, and our goal is to maximize the law of likelihood of the parameters of model.

Right. So we have some models for the joint distribution for X and Z and our goal is to find the maximum likeliness estimate of the parameters data where the likelihood is defined as something equals 1 to M [inaudible] probably of our data as usual. And here X is parameterized by data is now given by a sum over all the values of ZI parameterized by data. Okay. So just by taking our model of the joint distribution of X and Z and marginalizing out ZI that we get P of XI parameterized by data. And so the EM algorithm will be a way of performing this maximum likeliness estimation problem, which is complicated by the fact that we have these ZIs in our model that are unobserved. Before I actually do the math, here's a useful picture to

keep in mind. So the horizontal axis in this cartoon is the [inaudible] axis and there's some function, the law of likelihood of theta zero that we're trying to maximize, and usually maximizing our [inaudible] derivatives instead of the zero that would be very hard to do. What the EM algorithm will do is the following. Let's say it initializes some value of theta zero, what the EM algorithm will end up doing is it will construct a lower bound for this law of likelihood function and this lower bound will be tight [inaudible] of equality after current guessing the parameters and they maximize this lower boundary with respect to theta so we'll end up with say that value. So that will be data 1. Okay.

And then EM algorithm look at theta 1 and they'll construct a new lower bound of theta and then we'll maximize that. So you jump here. So that's the next theta 2 and you do that again and you get the same 3, 4, and so on until you converge to local optimum on [inaudible] theta function. Okay. So this is a cartoon that displays what the EM algorithm will do. So let's actually make that formal now. So you want to maximize with respect to theta sum of [inaudible] – there's my theta, so this is sum over 1 [inaudible] sum over all values of Z. Okay. So what I'm going to do is multiply and divide by the same thing and I'm gonna write this as Q – okay. So I'm going to construct the probability distribution QI, that would be over the latent random variables ZI and so these QI would get distribution so each of the QI would bring in a 0 and sum over all the values of ZI of QI would be 1, so these Qs will be a probability distribution that I get to construct. Okay. And then I'll later go describe the specific choice of this distribution QI. So this QI is a probability distribution over the random variables of ZI so this is [inaudible]. Right. I see some frowns. Do you have questions about this? No. Okay.

So the log function looks like that and there's a concave function so that tells us that the log of E of X is greater than and equal to the expected value of log X by the other concave function form of Jensen's and Equality. And so continuing from the previous expression, this is a summary of a log and an expectation, that must therefore be greater than or equal to the expected value of the log of that. Okay. Using Jensen's and Equality. And lastly just to expand out this formula again. This is equal to that. Okay. Yeah.

Student:
[Inaudible]
Instructor (Andrew Ng):
[Inaudible]
Student:
[Inaudible]. Yeah. Okay. So this has the [inaudible] so let's say

Random variable Z, right, and Z has some distribution. Let's denote it G. And let's say I have some function G of Z. Okay. Then by definition, the expected value of G of Z, by definition, that's equal to sum over all the values of Z, the probability of that value of Z times Z of G. Right. That's sort of the definition of a random variable. And so the way I got from this step to this step is by using that. So in particular, now, I've been using distribution QI to denote the distribution of Z, so this is, like, sum over Z of P of Z times [inaudible]. And so this is just the expected value with respect to a random variable Z joined from the distribution Q of G of Z. Are there questions?

Student: So in general when you're doing maximum likelihood estimations, the likelihood of the data, but in this case you only say probability of X because you only have observed X whereas previously we said probability of X given the labels?

Instructor (Andrew Ng):Yes. Exactly. Right. Right. [Inaudible] we want to choose the parameters that maximizes the probability of the data, and in this case, our data comprises only the Xs because we don't reserve the Zs, and therefore, the likelihood of parameters is given by the probability of the data, which is [inaudible]. So this is all we've done, right, we wanted to maximize the law of likelihood of theta and what we've done, through these manipulations, we've know constructed a lower bound on the law of likelihood of data. Okay.

And in particular, this formula that we came up, we should think of this as a function of theta then, if you think about it, theta are the parameters of your model, right, if you think about this as a function of your parameters theta, what we've just shown is that the law of likelihood of your parameters theta is lower bounded by this thing. Okay. Remember that cartoon of repeatedly constructing a lower bound and optimizing the lower bound. So what we've just done is construct a lower bound for the law of likelihood

for theta. Now, the last piece we want for this lower bound is actually we want this inequality to hold with equality for the current value for theta.

So just refrain back to the previous cartoon. If this was the law of likelihood for theta, we'd then construct some lower bound of it, some function of theta and if this is my current value for theta, then I want my lower bound to be tight. I want my lower bound to be equal to the law of likelihood of theta because that's what I need to guarantee that when I optimize my lower bound, then I'll actually do even better on the true objective function. Yeah.

Student:
How do [inaudible]

Instructor (Andrew Ng): Excuse me. Yeah. Great question. How do I know that function is concave? Yeah. I don't think I've shown it. It actually turns out to be true for all the models we work with. Do I know that the law of bound is a concave function of theta? I think you're right. In general, this may not be a concave function of theta. For many of the models we work with, this will turn out to be a concave function, but that's not always true. Okay. So let me go ahead and choose a value for Q. And I'll refer back to Jensen's and Equality. We said that this inequality will become an equality if the random variable inside is a constant. Right. If you're taking an expectation with respect to constant valued variables.

So the QI of ZIs must sum to 1 and so to compute it you should just take P of XI, ZI, parameterized by theta and just normalize the sum to one. There is a step that I'm skipping here to show that this is really the right thing to do. Hopefully, you'll just be convinced it's true. For the actual steps that I skipped, it's actually written out in the lecture notes. So you then have the denominator, by definition, is that and so by the definition of conditional probability QI of ZI is just equal to P of ZI given XI and parameterized by theta. Okay.

And so to summarize the algorithm, the EM algorithm has two steps. And the E step, we set, we choose the distributions QI, so QI of ZI will set to be equal to a P of ZI given [inaudible] by data. That's the formula we just worked out. And so by this step we've now created a lower bound on the law of likelihood function that is now tight at a current value of theta. And in the M step, we then optimize that lower bound with respect to our parameters theta and specifically to the [inaudible] of theta. Okay. And so that's the EM algorithm. I won't have time to do it today, but I'll probably show this in the next lecture, but the EM algorithm's that I wrote down for the mixtures of Gaussian's algorithm is actually a special case of this more general template where the E step and the M step responded. So pretty much exactly to this E step and this M step that I wrote down. The E step constructs this lower bound and makes sure that it is tight to the current value of theta. That's in my choice of Q, and then the M step optimizes the lower bound with respect to [inaudible] data. Okay. So lots more to say about this in the next lecture. Let's check if there's any questions before we close. No. Okay. Cool. So let's wrap up for today and we'll continue talking about this in the next session.

[End of Audio]
Duration: 76 minutes

1.13 Machine Learning Lecture 13[13]

Figure 1.13

MachineLearning-Lecture13

Instructor (Andrew Ng): Okay, good morning. For those of you actually online, sorry; starting a couple of minutes late. We're having trouble with the lights just now, so we're all sitting in the dark and they just came on. So welcome back, and what I want to do today is continue our discussions of the EM Algorithm, and in particular, I want to talk about the EM formulation that we derived in the previous lecture and apply it to the mixture of Gaussians model, apply it to a different model and a mixture of naive Bayes model, and then the launch part of today's lecture will be on the factor analysis algorithm, which will also use the EM. And as part of that, we'll actually take a brief digression to talk a little bit about sort of useful properties of Gaussian distributions.

So just to recap where we are. In the previous lecture, I started to talk about unsupervised learning, which was machine-learning problems, where you're given an unlabeled training set comprising m examples here, right? And then – so the fact that there are no labels; that's what makes this unsupervised or anything. So one problem that I talked about last time was what if you're given a data set that looks like this and you want to model the density PFX from which you think the data had been drawn, and so with a data set like this, maybe you think was a mixture of two Gaussians and start to talk about an algorithm for fitting a mixture of Gaussians model, all right? And so we said that we would model the density of XP of X as sum over Z PFX given Z times P of Z where this later random variable meaning this hidden random variable Z indicates which of the two Gaussian distributions each of your data points came from and so we have, you know, Z was not a nomial with parameter phi and X conditions on a coming from the JAFE Gaussian was given by Gaussian of mean mu J and covariant sigma J, all right?

So, like I said at the beginning of the previous lecture, I just talked about a very specific algorithm that I sort of pulled out of the air for fitting the parameters of this model for finian, Francis, phi, mu and sigma, but then in the second half of the previous lecture I talked about what's called the EM Algorithm in which our goal is that it's a likelihood estimation of parameters. So we want to maximize in terms of theta, you know, the, sort of, usual right matter of log likelihood – well, parameterized by theta. And because we have a later random variable Z this is really maximizing in terms of theta, sum over I, sum over Z, P of XI, ZI parameterized by theta. Okay? So using Jensen's inequality last time we worked out the EM Algorithm in which in the E step we would chose these probability distributions QI to the l posterior on Z given X and parameterized by theta and in the M step we would set theta to be the value that maximizes this. Okay? So these are the ones we worked out last time and the cartoon that I drew was that you have this long likelihood function L of theta that's often hard to maximize and what the E step does is choose these probability distribution production QI's. And in the cartoon, I drew what that corresponded to was finding a lower bounds for the log likelihood. And then horizontal access data and then the M step you maximize the lower boundary, right? So maybe you were here previously and so you jumped to the new point, the new maximum of this lower bound. Okay? And so this little curve here, right? This lower bound function here that's really the right-hand side of that augments. Okay? So this whole thing in the augments. If you view this thing as a function of theta, this function of theta is a lower bounds for the log likelihood of theta and so the M step we maximize this lower bound and that corresponds to jumping to this new maximum to lower bound.

So it turns out that in the EM Algorithm – so why do you evolve with the EM algorithm? It turns out that very often, and this will be true for all the examples we see today, it turns out that very often in the EM Algorithm maximizing the M Step, so performing the maximization the M Step, will be tractable and can often be done analytically in the closed form. Whereas if you were trying to maximize this objective we try to take this formula on the right and this maximum likely object, everyone, is to take this all on the right and set its derivatives to zero and try to solve and you'll find that you're unable to obtain a solution to this in closed form this maximization. Okay?

And so to give you an example of that is that you remember our discussion on exponential family marbles, right? It turns out that if X and Z is jointly, I guess, a line in exponential families. So if P of X, Z prioritized by theta there's an explanation family distribution, which it turns out to be true for the mixture of Gaussians distribution. Then turns out that the M step here will be tractable and the E step will also be tractable and so you can do each of these steps very easily. Whereas performing – trying to perform this original maximum likelihood estimation problem on this one, right? Will be computationally very difficult. You're going to set

the derivatives to zero and try to solve for that. Analytically you won't be able to find an analytic solution to this. Okay?

So what I want to do in a second is actually take this view of the EM Algorithm and apply it to the mixture of Gaussians models. I want to take these E steps and M Steps and work them out for the mixture of Gaussians model, but before I do that, I just want to say one more thing about this other view of the EM Algorithm. It turns out there's one other way of thinking about the EM Algorithm, which is the following: I can define an optimization objective J of theta, Q are defined it to be this. This is just a thing in the augments in the M step. Okay? And so what we proved using Jensen's inequality is that the log likelihood of theta is greater and equal to J of theta Q. So in other words, we proved last time that for any value of theta and Q the log likelihood upper bounds J of theta and Q. And so just to relate this back to, sort of, yet more things that you all ready know, you can also think of covariant cause in a sense, right? However, our discussion awhile back on the coordinate ascent optimization algorithm. So we can show, and I won't actually show this view so just take our word for it and look for that at home if you want, that EM is just coordinate in a set on the function J. So in the E step you maximize with respect to Q and then the M step you maximize with respect to theta. Okay? So this is another view of the EM Algorithm that shows why it has to converge, for example. If you can – I've used in a sense of J of theta, Q having to monotonically increase on every iteration. Okay?

So what I want to do next is actually take this general EM machinery that we worked up and apply it to a mixture Gaussians model. Before I do that, let me just check if there are questions about the EM Algorithm as a whole? Okay, cool.

So let's go ahead and work on the mixture of Gaussian's EM, all right? MOG, and that's my abbreviation for Mixture of Gaussian's. So the E step were called those Q distributions, right? In particular, I want to work out – so Q is the probability distribution over the late and random variable Z and so the E step I'm gonna figure out what is these compute – what is Q of ZI equals J. And you can think of this as my writing P of ZI equals J, right? Under the Q distribution. That's what this notation means. And so the EM Algorithm tells us that, let's see, Q of J is the likelihood probability of Z being the value J and given XI and all your parameters. And so, well, the way you compute this is by Dave's rule, right? So that is going to be equal to P of XI given ZI equals J times P of ZIJ divided by – right? That's all the – by Dave's rule. And so this you know because XI given ZI equals J. This was a Gaussian with mean mu J and covariant sigma J. And so to compute this first term you plug in the formula for the Gaussian density there with parameters mu J and sigma J and this you'd know because Z was not a nomial, right? Where parameters given by phi and so the problem of ZI being with J is just phi J and so you can substitute these terms in. Similarly do the same thing for the denominator and that's how you work out what Q is. Okay? And so in the previous lecture this value the probability that ZI equals J under the Q distribution that was why I denoted that as WIJ. So that would be the E step and then in the M step we maximize with respect to all of our parameters. This, well I seem to be writing the same formula down a lot today. All right. And just so we're completely concrete about how you do that, right? So if you do that you end up with – so plugging in the quantities that you know that becomes this, let's see. Right. And so that we're completely concrete about what the M step is doing. So in the M step that was, I guess, QI over Z, I being over J. Just in the summation, sum over J is the sum over all the possible values of ZI and then this thing here is my Gaussian density. Sorry, guys, this thing – well, this first term here, right? Is my P of XI given ZI and that's P of ZI. Okay? And so to maximize this with respect to – say you want to maximize this with respect to all of your parameters phi, mu and sigma. So to maximize with respect to the parameter mu, say, you would take the derivative for respect to mu and set that to zero and you would – and if you actually do that computation you would get, for instance, that that becomes your update to mu J. Okay? Just so I want to – the equation is unimportant. All of these equations are written down in the lecture notes. I'm writing these down just to be completely concrete about what the M step means. And so write down that formula, plug in the densities you know, take the derivative set to zero, solve for mu J and in the same way you set the derivatives equal to zero and solve for your updates for your other parameters phi and sigma as well. Okay?

Well, just point out just one little tricky bit for this that you haven't seen before that most of you probably all ready now, but I'll just mention is that since phi here is a multinomial distribution when you

take this formula and you maximize it with respect to phi you actually have an additional constraint, right? That the sum of I – let's see, sum over J, phi J must be equal to one. All right? So, again, in the M step I want to take this thing and maximize it with respect to all the parameters and when you maximize this respect to the parameters phi J you need to respect the constraint that sum of J phi J must be equal to one. And so, well, you all ready know how to do constraint maximization, right? So I'll throw out the method of the granjay multipliers and generalize the granjay when you talk about the support of X machines. And so to actually perform the maximization in terms of phi J you construct to the granjay, which is – all right? So that's the equation from above and we'll denote in the dot dot dot plus theta times that, where this is sort of the granjay multiplier and this is your optimization objective. And so to actually solve the parameters phi J you set the parameters of this so that the granjay is zero and solve. Okay? And if you then work through the math you get the appropriate value to update the phi J's too, which I won't do, but I'll be – all the full directions are in the lecture notes. I won't do that here.

Okay. And so if you actually perform all these computations you can also verify that. So I just wrote down a bunch of formulas for the EM Algorithm. At the beginning of the last lecture I said for the mixture of Gaussian's model – I said for the EM here's the formula for computing the WIJ's and here's a formula for computing the mud's and so on, and this variation is where all of those formulas actually come from. Okay? Questions about this? Yeah?

Student:[Inaudible]

Instructor (Andrew Ng): Oh, I see. So it turns out that, yes, there's also constrained to the phi J this must be greater than zero. It turns out that if you want you could actually write down then generalize the granjayn incorporating all of these constraints as well and you can solve to [inaudible] these constraints. It turns out that in this particular derivation – actually it turns out that very often we find maximum likely estimate for multinomial distributions probabilities. It turns out that if you ignore these constraints and you just maximize the formula luckily you end up with values that actually are greater than or equal to zero and so if even ignoring those constraint you end up with parameters that are greater than or equal to zero that shows that that must be the correct solution because adding that constraint won't change anything. So this constraint it is then caused – it turns out that if you ignore this and just do what I've wrote down you actually get the right answer. Okay? Great.

So let me just very quickly talk about one more example of a mixture model. And the perfect example for this is imagine you want to do text clustering, right? So someone gives you a large set of documents and you want to cluster them together into cohesive topics. I think I mentioned the news website news.google.com. That's one application of text clustering where you might want to look at all of the news stories about today, all the news stories written by everyone, written by all the online news websites about whatever happened yesterday and there will be many, many different stories on the same thing, right? And by running a text-clustering algorithm you can group related documents together. Okay?

So how do you apply the EM Algorithm to text clustering? I want to do this to illustrate an example in which you run the EM Algorithm on discreet valued inputs where the input – where the training examples XI are discreet values. So what I want to talk about specifically is the mixture of naïve Bayes model and depending on how much you remember about naïve Bayes I talked about two event models. One was the multivariant vanuvy event model. One was the multinomial event model. Today I'm gonna use the multivariant vanuvy event model. If you don't remember what those terms mean anymore don't worry about it. I think the equation will still make sense. But in this setting we're given a training set X1 for XM. So we're given M text documents where each XI is zero one to the end. So each of our training examples is an indimensional bit of vector, right? S this was the representation where XIJ was – it indicates whether word J appears in document I, right? And let's say that we're going to model ZI the – our latent random variable meaning our hidden random variable ZI will take on two values zero one so this means I'm just gonna find two clusters and you can generalize the clusters that you want.

So in the mixture of naïve Bayes model we assume that ZI is distributed and mu E with some value of phi so there's some probability of each document coming from cluster one or from cluster two. We assume that the probability of XI given ZI, right? Will make the same naïve Bayes assumption as we did before. Okay? And more specifically – well, excuse me, right. Okay. And so most of us [inaudible] cycles one given

ZI equals say zero will be given by a parameter phi substitute J given Z equals zero. So if you take this chalkboard and if you take all instances of the alphabet Z and replace it with Y then you end up with exactly the same equation as I've written down for naïve Bayes like a long time ago. Okay?

And I'm not actually going to work out the mechanics deriving the EM Algorithm, but it turns out that if you take this joint distribution of X and Z and if you work out what the equations are for the EM algorithm for maximum likelihood estimation of parameters you find that in the E step you compute, you know, let's say these parameters – these weights WI which are going to be equal to your perceived distribution of Z being equal one conditioned on XI parameterized by your phi's and given your parameters and in the M step. Okay? And that's the equation you get in the M step. I mean, again, the equations themselves aren't too important. Just sort of convey – I'll give you a second to finish writing, I guess. And when you're done or finished writing take a look at these equations and see if they make intuitive sense to you why these three equations, sort of, sound like they might be the right thing to do. Yeah?

Student:[Inaudible]

Instructor (Andrew Ng):Say that again.

Student:Y –

Instructor (Andrew Ng):Oh, yes, thank you. Right. Sorry, just, for everywhere over Y I meant Z. Yeah?

Student:[Inaudible] in the first place?

Instructor (Andrew Ng):No. So what is it? Normally you initialize phi's to be something else, say randomly. So just like in naïve Bayes we saw zero probabilities as a bad thing so the same reason you try to avoid zero probabilities, yeah. Okay? And so just the intuition behind these equations is in the E step WI's is you're gonna take your best guess for whether the document came from cluster one or cluster zero, all right? This is very similar to the intuitions behind the EM Algorithm that we talked about in a previous lecture. So in the E step we're going to compute these weights that tell us do I think this document came from cluster one or cluster zero. And then in the M step I'm gonna say does this numerator is the sum over all the elements of my training set of – so then informally, right? WI is one there, but I think the document came from cluster one and so this will essentially sum up all the times I saw words J in documents that I think are in cluster one. And these are sort of weighted by the actual probability. I think it came from cluster one and then I'll divide by – again, if all of these were ones and zeros then I'd be dividing by the actual number of documents I had in cluster one. So if all the WI's were either ones or zeroes then this would be exactly the fraction of documents that I saw in cluster one in which I also saw were at J. Okay? But in the EM Algorithm you don't make a hard assignment decision about is this in cluster one or is this in cluster zero. You instead represent your uncertainty about cluster membership with the parameters WI. Okay?

It actually turns out that when we actually implement this particular model it actually turns out that by the nature of this computation all the values of WI's will be very close to either one or zero so they'll be numerically almost indistinguishable from one's and zeroes. This is a property of naïve Bayes. If you actually compute this probability from all those documents you find that WI is either 0.0001 or 0.999. It'll be amazingly close to either zero or one and so the M step – and so this is pretty much guessing whether each document is in cluster one or cluster zero and then using formulas they're very similar to maximum likely estimation for naïve Bayes. Okay? Cool. Are there – and if some of these equations don't look that familiar to you anymore, sort of, go back and take another look at what you saw in naïve Bayes and hopefully you can see the links there as well. Questions about this before I move on? Right, okay.

Of course the way I got these equations was by turning through the machinery of the EM Algorithm, right? I didn't just write these out of thin air. The way you do this is by writing down the E step and the M step for this model and then the M step same derivatives equal to zero and solving from that so that's how you get the M step and the E step.

So the last thing I want to do today is talk about the factor analysis model and the reason I want to do this is sort of two reasons because one is factor analysis is kind of a useful model. It's not as widely used as mixtures of Gaussian's and mixtures of naïve Bayes maybe, but it's sort of useful. But the other reason I want to derive this model is that there are a few steps in the math that are more generally useful.

In particular, where this is for factor analysis this would be an example in which we'll do EM where the late and random variable – where the hidden random variable Z is going to be continued as valued. And so some of the math we'll see in deriving factor analysis will be a little bit different than what you saw before and they're just a – it turns out the full derivation for EM for factor analysis is sort of extremely long and complicated and so I won't inflect that on you in lecture today, but I will still be writing more equations than is – than you'll see me do in other lectures because there are, sort of, just a few steps in the factor analysis derivation so I'll physically illustrate it.

So it's actually [inaudible] the model and it's really contrast to the mixture of Gaussians model, all right? So for the mixture of Gaussians model, which is our first model we had, that – well I actually motivated it by drawing the data set like this, right? That one of you has a data set that looks like this, right? So this was a problem where n is two-dimensional and you have, I don't know, maybe 50 or 100 training examples, whatever, right? And I said maybe you want to give a label training set like this. Maybe you want to model this as a mixture of two Gaussians. Okay? And so a mixture of Gaussian models tend to be applicable where m is larger, and often much larger, than n where the number of training examples you have is at least as large as, and is usually much larger than, the dimension of the data.

What I want to do is talk about a different problem where I want you to imagine what happens if either the dimension of your data is roughly equal to the number of examples you have or maybe the dimension of your data is maybe even much larger than the number of training examples you have. Okay? So how do you model such a very high dimensional data? Watch and you will see sometimes, right? If you run a plant or something, you run a factory, maybe you have a thousand measurements all through your plants, but you only have five – you only have 20 days of data. So you can have 1,000 dimensional data, but 20 examples of it all ready. So given data that has this property in the beginning that we've given a training set of m examples. Well, what can you do to try to model the density of X? So one thing you can do is try to model it just as a single Gaussian, right? So in my mixtures of Gaussian this is how you try model as a single Gaussian and say X is intuitive with mean mu and parameter sigma where sigma is going to be done n by n matrix and so if you work out the maximum likelihood estimate of the parameters you find that the maximum likelihood estimate for the mean is just the empirical mean of your training set, right. So that makes sense. And the maximum likelihood of the covariance matrix sigma will be this, all right? But it turns out that in this regime where the data is much higher dimensional – excuse me, where the data's dimension is much larger than the training examples you have if you compute the maximum likely estimate of the covariance matrix sigma you find that this matrix is singular. Okay? By singular, I mean that it doesn't have four vanq or it has zero eigen value so it doesn't have – I hope one of those terms makes sense. And there's another saying that the matrix sigma will be non-invertible. And just in pictures, one complete example is if D is – if N equals M equals two if you have two-dimensional data and you have two examples. So I'd have two training examples in two-dimen – this is X1 and X2. This is my unlabeled data. If you fit a Gaussian to this data set you find that – well you remember I used to draw constables of Gaussians as ellipses, right? So these are examples of different constables of Gaussians. You find that the maximum likely estimate Gaussian for this responds to Gaussian where the contours are sort of infinitely thin and infinitely long in that direction. Okay? So in terms – so the contours will sort of be infinitely thin, right? And stretch infinitely long in that direction. And another way of saying it is that if you actually plug in the formula for the density of the Gaussian, which is this, you won't actually get a nice answer because the matrix sigma is non-invertible so sigma inverse is not defined and this is zero. So you also have one over zero times E to the sum inversive and non-inversive matrix so not a good model. So let's do even better, right? So given this sort of data how do you model P of X?

Well, one thing you could do is constrain sigma to be diagonal. So you have a covariance matrix X is – okay? So in other words you get a constraint sigma to be this matrix, all right? With zeroes on the off diagonals. I hope this makes sense. These zeroes I've written down here denote that everything after diagonal of this matrix is a zero. So the massive likely estimate of the parameters will be pretty much what you'll expect, right? And in pictures what this means is that the [inaudible] the distribution with Gaussians whose controls are axis aligned. So that's one example of a Gaussian where the covariance is diagonal. And here's another example and so here's a third example. But often I've used the examples of Gaussians where

the covariance matrix is off diagonal. Okay? And, I don't know, you could do this in model P of X, but this isn't very nice because you've now thrown away all the correlations between the different variables so the axis are X1 and X2, right? So you've thrown away – you're failing to capture any of the correlations or the relationships between any pair of variables in your data. Yeah?

Student: Is it – could you say again what does that do for the diagonal?

Instructor (Andrew Ng): Say again.

Student: The covariance matrix the diagonal, what does that again? I didn't quite understand what the examples mean.

Instructor (Andrew Ng): Okay. So these are the contours of the Gaussian density that I'm drawing, right? So let's see – so post covariance issues with diagonal then you can ask what is P of X parameterized by mu and sigma, right? If sigma is diagonal and so this will be some Gaussian dump, right? So not in – oh, boy. My drawing's really bad, but in two-D the density for Gaussian is like this bump shaped thing, right? So this is the density of the Gaussian – wow, and this is a really bad drawing. With those, your axis X1 and X2 and the height of this is P of X and so those figures over there are the contours of the density of the Gaussian. So those are the contours of this shape.

Student: No, I don't mean the contour. What's special about these types? What makes them different than instead of general covariance matrix?

Instructor (Andrew Ng): Oh, I see. Oh, okay, sorry. They're axis aligned so the main – these, let's see. So I'm not drawing a contour like this, right? Because the main axes of these are not aligned with the X1 and X-axis so this occurs found to Gaussian where the off-diagonals are non-zero, right? Cool. Okay. You could do this, this is sort of work. It turns out that what our best view is two training examples you can learn in non-singular covariance matrix, but you've thrown away all of the correlation in the data so this is not a great model.

It turns out you can do something – well, actually, we'll come back and use this property later. But it turns out you can do something even more restrictive, which is you can constrain sigma to equal to sigma squared times the identity matrix. So in other words, you can constrain it to be diagonal matrix and moreover all the diagonal entries must be the same and so the cartoon for that is that you're constraining the contours of your Gaussian density to be circular. Okay? This is a sort of even harsher constraint to place in your model. So either of these versions, diagonal sigma or sigma being the, sort of, constant value diagonal are the all ready strong assumptions, all right? So if you have enough data maybe write a model just a little bit of a correlation between your different variables. So the factor analysis model is one way to attempt to do that. So here's the idea. So this is how the factor analysis model models your data. We're going to assume that there is a latent random variable, okay? Which just means random variable Z. So Z is distributed Gaussian with mean zero and covariance identity where Z will be a D-dimensional vector now and D will be chosen so that it is lower than the dimension of your X's. Okay? And now I'm going to assume that X is given by – well let me write this. Each XI is distributed – actually, sorry, I'm just. We have to assume that conditions on the value of Z, X is given by another Gaussian with mean given by mu plus lambda Z and covariance given by matrix si. So just to say the second line in an equivalent form, equivalently I'm going to model X as mu plus lambda Z plus a noise term epsilon where epsilon is Gaussian with mean zero and covariant si. And so the parameters of this model are going to be a vector mu with its n-dimensional and matrix lambda, which is n by D and a covariance matrix si, which is n by n, and I'm going to impose an additional constraint on si. I'm going to impose a constraint that si is diagonal. Okay? So that was a form of definition – let me actually, sort of, give a couple of examples to make this more complete. So let's give a kind of example, suppose Z is one-dimensional and X is two-dimensional so let's see what this model – let's see a, sort of, specific instance of the factor analysis model and how we're modeling the joint – the distribution over X of – what this gives us in terms of a model over P of X, all right?

So let's see. From this model to let me assume that lambda is 2, 1 and si, which has to be diagonal matrix, remember, is this. Okay? So Z is one-dimensional so let me just draw a typical sample for Z, all right? So if I draw ZI from a Gaussian so that's a typical sample for what Z might look like and so I'm gonna – at any rate I'm gonna call this Z1, Z2, Z3 and so on. If this really were a typical sample the order of the Z's would be jumbled up, but I'm just ordering them like this just to make the example easier. So,

yes, typical sample of random variable Z from a Gaussian distribution with mean of covariance one. So – and with this example let me just set mu equals zero. It's to write the – just that it's easier to talk about. So lambda times Z, right? We'll take each of these numbers and multiply them by lambda. And so you find that all of the values for lambda times Z will lie on a straight line, all right? So, for example, this one here would be one, two, three, four, five, six, seven, I guess. So if this was Z7 then this one here would be lambda times Z7 and now that's the number in R2, because lambda's a two by one matrix. And so what I've drawn here is like a typical sample for lambda times Z and the final step for this is what a typical sample for X looks like. Well X is mu plus lambda Z plus epsilon where epsilon is Gaussian with mean nu and covariance given by si, right? And so the last step to draw a typical sample for the random variables X I'm gonna take these non – these are really same as mu plus lambda Z because mu is zero in this example and around this point I'm going to place an axis aligned ellipse. Or in other words, I'm going to create a Gaussian distribution centered on this point and this I've drawn corresponds to one of the contours of my density for epsilon, right? And so you can imagine placing a little Gaussian bump here. And so I'll draw an example from this little Gaussian and let's say I get that point going, I do the same here and so on. So I draw a bunch of examples from these Gaussians and the – whatever they call it – the orange points I drew will comprise a typical sample for whether distribution of X is under this model. Okay? Yeah?

Student: Would you add, like, mean? Instructor:

Oh, say that again.

Student: Do you add mean into that?

Instructor (Andrew Ng): Oh, yes, you do. And in this example, I said you do a zero zero just to make it easier. If mu were something else you'd take the whole picture and you'd sort of shift it to whatever value of mu is. Yeah?

Student:[Inaudible] horizontal line right there, which was Z. What did the X's, of course, what does that Y-axis corresponds to?

Instructor (Andrew Ng): Oh, so this is Z is one-dimensional so here I'm plotting the typical sample for Z so this is like zero. So this is just the Z Axis, right. So Z is one-dimensional data. So this line here is like a plot of a typical sample of values for Z. Okay? Yeah?

Student: You have by axis, right? And the axis data pertains samples.

Instructor (Andrew Ng): Oh, yes, right.

Student: So sort of projecting them into that?

Instructor (Andrew Ng): Let's not talk about projections yet, but, yeah, right. So these beige points – so that's like X1 and that's X2 and so on, right? So the beige points are what I see. And so in reality all you ever get to see are the X's, but just like in the mixture of Gaussians model I tell a story about what I would imagine the Gau—the data came from two Gaussian's was is had a random variable Z that led to the generation of X's from two Gaussians. So the same way I'm sort of telling the story here, which all the algorithm actually sees are the orange points, but we're gonna tell a story about how the data came about and that story is what comprises the factor analysis model. Okay? So one of the ways to see the intrusion of this model is that we're going to think of the model as one way just informally, not formally, but one way to think about this model is you can think of this factor analysis model as modeling the data from coming from a lower dimensional subspace more or less so the data X here Y is approximately on one D line and then plus a little bit of noise – plus a little bit of random noise so the X isn't exactly on this one D line. That's one informal way of thinking about factor analysis.

We're not doing great on time. Well, let's do this. So let me just do one more quick example, which is, in this example, let's say Z is in R2 and X is in R3, right? And so in this example Z, your data Z now lies in 2-D and so let me draw this on a sheet of paper. Okay? So let's say the axis of my paper are the Z1 and Z2 axis and so here is a typical sample of point Z, right? And so we'll then take the sample Z – well, actually let me draw this here as well. All right. So this is a typical sample for Z going on the Z1 and Z2 axis and I guess the origin would be here. So center around zero. And then we'll take those and map it to mu plus lambda Z and what that means is if you imagine the free space of this classroom is R3. What that means is we'll take this sample of Z's and we'll map it to position in free space. So we'll take this sheet of paper and move it somewhere and some orientation in 3-D space. And the last step is you have X equals

mu plus lambda Z plus epsilon and so you would take the set of the points which align in some plane in our 3-D space the variable of noise of these and the noise will, sort of, come from Gaussians to the axis aligned. Okay? So you end up with a data set that's sort of like a fat pancake or a little bit of fuzz off your pancake. So that's a model – let's actually talk about how to fit the parameters of the model. Okay?

In order to describe how to fit the model I'm sure we need to re-write Gaussians and this is in a very slightly different way. So, in particular, let's say I have a vector X and I'm gonna use this notation to denote partition vectors, right? X1, X2 where if X1 is say an r-dimensional vector then X2 is an estimational vector and X is an R plus S dimensional vector. Okay? So I'm gonna use this notation to denote just the taking of vector and, sort of, partitioning the vector into two halves. The first R elements followed by the last S elements. So let's say you have X coming from a Gaussian distribution with mean mu and covariance sigma where mu is itself a partition vector. So break mu up into two pieces mu1 and mu2 and the covariance matrix sigma is now a partitioned matrix. Okay? So what this means is that you take the covariance matrix sigma and I'm going to break it up into four blocks, right? And so the dimension of this is there will be R elements here and there will be S elements here and there will be R elements here. So, for example, sigma 1, 2 will be an R by S matrix. It's R elements tall and S elements wide.

So this Gaussian over to down is really a joint distribution of a loss of variables, right? So X is a vector so XY is a joint distribution over X1 through X of – over XN or over X of R plus S. We can then ask what are the marginal and conditional distributions of this Gaussian? So, for example, with my Gaussian, I know what P of X is, but can I compute the modular distribution of X1, right. And so P of X1 is just equal to, of course, integrate our X2, P of X1 comma X2 DX2. And if you actually perform that distribution – that computation you find that P of X1, I guess, is Gaussian with mean given by mu1 and sigma 1, 1. All right. So this is sort of no surprise. The marginal distribution of a Gaussian is itself the Gaussian and you just take out the relevant sub-blocks of the covariance matrix and the relevant sub-vector of the mu vector – E in vector mu. You can also compute conditionals. You can also – what does P of X1 given a specific value for X2, right? And so the way you compute that is, well, the usual way P of X1 comma X2 divided by P of X2, right? And so you know what both of these formulas are, right? The numerator – well, this is just a usual Gaussian that your joint distribution over X1, X2 is a Gaussian with mean mu and covariance sigma and this by that marginalization operation I talked about is that. So if you actually plug in the formulas for these two Gaussians and if you simplify the simplification step is actually fairly non-trivial. If you haven't seen it before this will actually be – this will actually be somewhat difficult to do. But if you plug this in for Gaussian and simplify that expression you find that conditioned on the value of X2, X1 is – the distribution of X1 conditioned on X2 is itself going to be Gaussian and it will have mean mu of 1 given 2 and covariant sigma of 1 given 2 where – well, so about the simplification and derivation I'm not gonna show the formula for mu given – of mu of one given 2 is given by this and I think the sigma of 1 given 2 is given by that. Okay?

So these are just useful formulas to know for how to find the conditional distributions of the Gaussian and the marginal distributions of a Gaussian. I won't actually show the derivation for this.

Student: Could you repeat the [inaudible]?

Instructor (Andrew Ng): Sure. So this one on the left mu of 1 given 2 equals mu1 plus sigma 1,2, sigma 2,2 inverse times X2 minus mu2 and this is sigma 1 given 2 equals sigma 1,1 minus sigma 1,2 sigma 2,2 inverse sigma 2,1. Okay? These are also in the lecture notes. Shoot. Nothing as where I was hoping to on time. Well, actually it is. Okay?

So it turns out – I think I'll skip this in the interest of time. So it turns out that – well, so let's go back and use these in the factor analysis model, right? It turns out that you can go back and – oh, do I want to do this? I kind of need this though. So let's go back and figure out just what the joint distribution factor analysis assumes on Z and X's. Okay? So under the factor analysis model Z and X, the random variables Z and X have some joint distribution given by – I'll write this vector as mu ZX in some covariance matrix sigma. So let's go back and figure out what mu ZX is and what sigma is and I'll do this so that we'll get a little bit more practice with partition vectors and partition matrixes. So just to remind you, right? You have to have Z as Gaussian with mean zero and covariance identity and X is mu plus lambda Z plus epsilon where epsilon is Gaussian with mean zero covariant si. So I have the – I'm just writing out the same

equations again. So let's first figure out what this vector mu ZX is. Well, the expected value of Z is zero and, again, as usual I'll often drop the square backers around here. And the expected value of X is – well, the expected value of mu plus lambda Z plus epsilon. So these two terms have zero expectation and so the expected value of X is just mu and so that vector mu ZX, right, in my parameter for the Gaussian this is going to be the expected value of this partition vector given by this partition Z and X and so that would just be zero followed by mu. Okay? And so that's a d-dimensional zero followed by an indimensional mu. That's not gonna work out what the covariance matrix sigma is. So the covariance matrix sigma – if you work out definition of a partition. So this is into your partition matrix. Okay? Will be – so the covariance matrix sigma will comprise four blocks like that and so the upper left most block, which I write as sigma 1,1 – well, that uppermost left block is just the covariance matrix of Z, which we know is the identity. I was gonna show you briefly how to derive some of the other blocks, right, so sigma 1,2 that's the upper – oh, actually, excuse me. Sigma 2,1 which is the lower left block that's E of X minus EX times Z minus EZ. So X is equal to mu plus lambda Z plus epsilon and then minus EX is minus mu and then times Z because the expected value of Z is zero, right, so that's equal to zero. And so if you simplify – or if you expand this out plus mu minus mu cancel out and so you have the expected value of lambda – oh, excuse me. ZZ transpose minus the expected value of epsilon Z is equal to that, which is just equal to lambda times the identity matrix. Okay? Does that make sense? Cause this term is equal to zero. Both epsilon and Z are independent and have zero expectation so the second terms are zero.

Well, so the final block is sigma 2,2 which is equal to the expected value of mu plus lambda Z plus epsilon minus mu times, right? Is equal to – and I won't do this, but this simplifies to lambda lambda transpose plus si. Okay? So putting all this together this tells us that the joint distribution of this vector ZX is going to be Gaussian with mean vector given by that, which we worked out previously. So this is the new ZX that we worked out previously, and covariance matrix given by that. Okay? So in principle – let's see, so the parameters of our model are mu, lambda, and si. And so in order to find the parameters of this model we're given a training set of m examples and so we like to do a massive likely estimation of the parameters. And so in principle one thing you could do is you can actually write down what P of XI is and, right, so P of XI XI is actually – the distribution of X, right? If, again, you can marginalize this Gaussian and so the distribution of X, which is the lower half of this partition vector is going to have mean mu and covariance given by lambda lambda transpose plus si. Right? So that's the distribution that we're using to model P of X.

And so in principle one thing you could do is actually write down the log likelihood of your parameters, right? Which is just the product over of – it is the sum over I log P of XI where P of XI will be given by this Gaussian density, right. And I'm using theta as a shorthand to denote all of my parameters. And so you actually know what the density for Gaussian is and so you can say P of XI is this Gaussian with E mu in covariance given by this lambda lambda transpose plus si. So in case you write down the log likelihood of your parameters as follows and you can try to take derivatives of your log likelihood with respect to your parameters and maximize the log likelihood, all right. It turns out that if you do that you end up with sort of an intractable atomization problem or at least one that you – excuse me, you end up with a optimization problem that you will not be able to find and in this analytics, sort of, closed form solutions to. So if you say my model of X is this and found your massive likely parameter estimation you won't be able to find the massive likely estimate of the parameters in closed form. So what I would have liked to do is – well, so in order to fit parameters to this model what we'll actually do is use the EM Algorithm in with the E step, right? We'll compute that and this formula looks the same except that one difference is that now Z is a continuous random variable and so in the E step we actually have to find the density QI of ZI where it's the, sort of, E step actually requires that we find the posterior distribution that – so the density to the random variable ZI and then the M step will then perform the following maximization where, again, because Z is now continuous we now need to integrate over Z. Okay? Where in the M step now because ZI was continuous we now have an integral over Z rather than a sum. Okay?

I was hoping to go a little bit further in deriving these things, but I don't have time today so we'll wrap that up in the next lecture, but before I close let's check if there are questions about the whole factor analysis model. Okay. So we'll come back in the next lecture; I will wrap up this model and because I want to go a

little bit deeper into the E and M steps, as there's some tricky parts for the factor analysis model specifically. Okay. I'll see you in a couple of days.

[End of Audio]

Duration: 75 minutes

1.14 Machine Learning Lecture 14[14]

Figure 1.14

MachineLearning-Lecture14

Instructor (Andrew Ng): All right. Good morning. Just a couple quick announcements before I get started. One is you should have seen Ziko's e-mail yesterday already. Several of you had asked me about – let you know [inaudible], so I wrote those up. We posted them online yesterday. The syllabus for the midterm is everything up to and including last Wednesday's lecture, so I guess [inaudible] is on the syllabus. You can take at the notes if you want. And also practice midterm had been posted on the course website, so you can take a look at that, too. The midterm will be in Terman auditorium tomorrow at 6:00 p.m. Directions were sort of included – or links to directions were included in Ziko's e-mail. And we actually at 6:00 p.m. sharp tomorrow, so do come a little bit before 6:00 p.m. to make sure you're seated by 6:00 p.m. as we'll hand out the midterms a few minutes before 6:00 p.m. and we'll start the midterm at 6:00 p.m. Okay?

Are there any questions about midterms? Any logistical things? Are you guys excited? Are you looking forward to the midterm? All right. Okay. So welcome back, and what I want to do to is talk about – is wrap up our discussion on factor analysis, and in particular what I want to do is step through parts of the derivations for EM for factor analysis because again there are a few steps in the EM derivation that are particularly tricky, and there are specific mistakes that people often make on deriving EM algorithms for algorithms like factor analysis. So I wanted to show you how to do those steps right so you can apply the same ideas to other problems as well. And then in the second half or so of this lecture, I'll talk about principal component analysis, which is a very powerful algorithm for dimensionality reduction. We'll see later what that means.

So just a recap, in a previous lecture I described a few properties of Gaussian distributions. One was that if you have a random variable – a random value vector X that can be partitioned into two portions, X1 and X2, and if X is Gaussian with mu [inaudible] and covariance sigma where mu is itself a partition vector and sigma is sort of a partition matrix that can be written like that. So I'm just writing sigma in terms of the four sub-blocks. Then you can look at the distribution of X and ask what is the marginal distribution of say X1. And the answer we said last time was that X1 – the marginal distribution of X1 is Gaussian would mean mu and covariance sigma one one, whereas sigma one one is the upper left block of that covariance matrix sigma. So this one is no surprise.

And I also wrote down the formula for computing conditional distributions, such as what is P of X1 given X2, and last time I wrote down that the distribution of X1 given X2 would also be Gaussian with parameters that I wrote as mu of one given two and sigma of one given two where mu of one given two is – let's see [inaudible] this formula. Okay? So with these formulas will be able to locate a pair of joint Gaussian random variables – X1 and X here are both vectors – and compute the marginal and conditional distributions, so P of X1 or P of X1 given X2. So when I come back and derive the E set – actually, I'll come back and use the marginal formula in a second, and then when I come back and derive from the E step in the EM algorithm for factor analysis, I'll actually be using these two formulas again.

And again, just to continue summarizing what we did last time, we said that in factor analysis our model – let's see. This is an unsupervised learning problem, and so we're given an unlabeled training set where each SI is a vector in RN as usual. We want to model the density of X, and our model for X would be that we imagine there's a latent random variable Z that's generating this [inaudible] zero mean in identity covariance. And Z will be some low dimensional thing [inaudible] and we [inaudible]. And we imagine that X is generated as mu plus lambda Z plus epsilon where epsilon is a Gaussian random variable with mean zero and a covariance matrix psi. And so the parameters of this model are mu which N-dimensional vector, lambda, which is an N by D-dimensional vector – matrix, and psi which is N by N and is diagonal. N is a diagonal matrix. So the cartoon I drew last time for factor analysis was – I said that maybe that's the typical example of data point ZI if – and in this example I had D equals one, N equals two. So Z in this example is one-dimensional. D equals one. And so you take this data, map it to say [inaudible] mu plus lambda Z and that may give you some set of points there.

And lastly, this model was envisioning that you'd place a little Gaussian bump around each of these say and sample – and the Xs are then maybe – that would be a typical sample of the Xs under this model. So how [inaudible] the parameters of this model? Well, the joint distribution of Z and X is actually Gaussian where parameters given by some vector [inaudible] mu XZ, and sum covariance matrix sigma. And [inaudible] what those two things are, this vector [inaudible] is a vector of zeroes appended to the vector mu. And the matrix sigma is this partitioned matrix.

We also worked this out last time. So you can ask what is the distribution of X under this model, and the answer is under this model X is Gaussian with mean mu and covariance lambda lambda [inaudible] plus psi. So let's just take the second block of the mean vector and take that lower right hand corner block for the covariance matrix, and so this is really my formula for computing the marginal distribution of a Gaussian, except that I'm computing the marginal distribution of the second half of the vector rather than the first half. So this is the marginal distribution of X under my model. And so if you want to learn –

Student:[Inaudible] initial distribution [inaudible]?

Instructor (Andrew Ng):Let's see. Oh, yes. Yes, so in this one I'm breaking down the – this is really I'm specifying the conditional distribution of X given Z. So the conditional distribution of X given Z – this is Gaussian – would mean mu plus lambda Z and covariance psi. This is what that [inaudible]. So since this is the marginal distribution of X, given my training set of M unlabeled examples, I can actually write down the log likelihood of my training set. So the log likelihood of my training set – actually, no. Let's just write down the likelihood. So the likelihood of my parameters given my training set is the product from I equals one to M of P of XI given the parameters.

I can actually write down what that is because XI is Gaussian with parameter mu and variance lambda lambda transpose times psi, so you can actually write this down as one over two pi to the N over two determined of – and then times E to the minus one half X of – so that's my formula for the density of a Gaussian that has mean mu and covariance lambda lambda transpose plus psi. So this is my likelihood of the parameters given a training set. And one thing you could do is actually take this likelihood and try to maximize it in terms of the parameters, try to find the [inaudible] the parameters. But if you do that you find that – if you sort of try – take [inaudible] to get the law of likelihood, take derivatives, set derivatives equal to zero, you find that you'll be unable to solve for the maximum of this analytically. You won't be able to solve this [inaudible] likelihood estimation problem.

If you take the derivative of this with respect to the parameters, set the derivatives equal to zero and try to solve for the value of the parameters lambda, mu, and psi [inaudible] so you won't be able to do that [inaudible]. So what we'll use instead to estimate the parameters in a factor analysis model will be the EM algorithm.

Student: Why is the law of likelihood P of X and not P of X [inaudible] X and Z or something?

Instructor (Andrew Ng):Oh, right. So the question is why is the likelihood P of X and not P of X given Z or P of X and Z. The answer is let's see – we're in the – so by analogy to the mixture of Gaussian distributions models, we're given a training set that just comprises a set of unlabeled training examples that – for convenience for whatever reason, it's easier for me to write down a model that defines the joint distribution on P of X comma Z.

But what I would like to do is really maximize – as a function of my parameters – I'm using theta as a shorthand to denote all the parameters of the model – I want to maximize the probability of the data I actually observe. And this would be actually [inaudible] theta [inaudible] from I equals one to M, and this is really integrating out Z.

So I only ever get to observe the Xs and the Zs are latent random variables or hidden random variables. And so what I'd like to do is do this maximization in which I've implicitly integrated out Z. Does that make sense? Actually, are there other questions about the factor analysis models? Okay. So here's the EM algorithm. In the E step, we compute the conditional distribution of ZI given XI in our current setting of the parameters. And in the M step, we perform this maximization. And if this looks a little bit different from the previous versions of EM that you've seen, the only difference is that now I'm integrating over ZI because ZI is now a Gaussian random variable, is now this continuous value thing, so rather than summing over ZI, I now integrate over ZI. And if you replace [inaudible] of a sum, you [inaudible] ZI then this is exactly the M step you saw when we worked it out from the mixture of Gaussian [inaudible].

So it turns out that in order to implement the E step and the M step, there are just two little things – there are actually sort of three key things that are different from the models that you saw previously, so I wanna talk about them. The first one is that the first of the [inaudible] which is that in the E step, Z is now a continuous value random variable, so you now need a way to represent these continuous value densities, probably density functions to represent QI of Z. So fortunately in this probably it's not difficult to do, and in particular the conditional distribution of ZI given XI and our parameters which I'm gonna omit from this equation is going to be Gaussian with mean and covariance given by these two things, so I write like that, where this is going to be equal to the vector zero minus –

And the way I got this formula was – if you match this to the formula I had previously for computing conditional distributions of Gaussians, this corresponded to the terms mu one minus sigma one two sigma two two inverse times two minus mu. So those are the terms corresponding to the form I had previously for computing the marginal distributions of a Gaussian. And that is going to be given by that, and again those are the terms corresponding to the formulas I had for the very first thing I did, the formulas for computing conditional distributions of Gaussians. And so this is E step. You need to compute the Q distribution. You need to compute QI of ZI, and to do that what you actually do is you compute this vector mu of ZI given XI and sigma of ZI given XI, and together these represent the mean and covariance of the distribution Q where Q is going to be Gaussian, so that's the E step. Now here's the M step then. I'll just mention there are sort of two ways to derive M steps for especially Gaussian models like these. Here's the key trick I guess which is that when you compute the M step, you often need to compute integrals that look like these. And then there'll be some function of ZI. Let me just write ZI there, for instance.

So there are two ways to compute integrals like these. One is if you write out – this is a commonly made mistake in – well, not mistake, commonly made unnecessary complication I guess. One way to try to compute this integral is you can write this out as integral over ZI. And while we know what QI is, right? QI is a Gaussian so two pi, so D over two – the covariance of QI is this sigma given XI which you've computed in the E step, and then times E to the minus one half ZI minus mu of ZI given XI transpose sigma inverse – so that's my Gaussian density. And then times ZI ZZI. And so writing those out is the unnecessarily complicated way to do it because once you've written out this integral, if you want to actually integrate – that's the times, multiplication – if you actually want to evaluate this integral, it just looks horribly complicated. I'm not quite sure how to evaluate this. By far, the simpler to evaluate this integral is to recognize that this is just the expectation with respect to ZI drawn from the distribution QI of the random variable ZI. And once you've actually got this way, you notice that this is just the expected value of ZI under the distribution QI, but QI is a Gaussian distribution with mean given by that mu vector and covariance given by that sigma vector, and so the expected value of ZI under this distribution is just mu of ZI given XI. Does that make sense? So by making those observations that this is just an expected value, there's a much easier way to compute that integral. [Inaudible]. So we'll apply the same idea to the M step. So the M step we want to maximize this. There's also sum over I – there's a summation over I outside, but this is essentially the term inside the arg max of the M step where taking the integral of a ZI and just observe that that's actually – I guess just form some expectation respect to the random variable ZI of this thing inside. And so this

simplifies to the following.

And it turns out actually all I did here was use the fact that P of X given Z times P of Z equals P over X comma Z. Right? That's just combining this term and that term gives you the numerator in the original. And so in turns out that for factor analysis in these two terms, this is the only one that depends on the parameters. The distribution P of ZI has no parameters because ZI was just drawn from a Gaussian with zero mean and identity covariance. QI of Z was this fixed Gaussian. It doesn't depend on the parameters theta, and so in the M step we really just need to maximize this one term with respect to all parameters, mu, lambda, and psi. So let's see. There's sort of one more key step I want to show but to get there I have to write down an unfortunately large amount of math, and let's go into it. Okay.

So in the M step, we want to maximize and all expectations with respect to ZI drawn from the distributions QI, and sometimes I'll be sloppy and just omit this. And now this distribution P of XI given ZI, that is a Gaussian density because XI given ZI – this is Gaussian with mean given by mu plus lambda Z and covariance psi. And so in this step of the derivation, I will actually go ahead and substitute in the formula for Gaussian density. So I will go ahead and take those and plug it into here, one over two pi to the N over two, psi inverse E to the dot, dot, dot. So I will go ahead and plug in the Gaussian density. And when you do that, you find that you get expectation – excuse me. I forgot to say to maintain – to not make the derivation too complicated, I'm actually just going to maximize this with respect to the parameters lambda. So let me just show how the – so you want to maximize this with respect to lambda, psi, and mu, but just to keep the amount of math I'm gonna do in class sane, I'm just going to show how to maximize this with respect to the matrix lambda, and I'll pretend that psi and mu are fixed. And so if you substitute in the Gaussian density, you get some expected value of the constant. The constant may depend on psi but not on lambda. Then mine is this thing. And this quadratic term essentially came from the exponent in your Gaussian density. When I take log of exponent, then you end up with this quadratic term.

And so if you take the derivatives of the expression above with respect to the matrix lambda and you set that to zero – we want to maximize this expression with respect to the parameters lambda, so you take the derivative of this with respect to lambda – excuse me. That's a minus sign – and set the derivate of this expression to zero because you set derivatives to zero to maximize things, right? When you do that and simplify, you end up with the following. And so that's the – in the M step, this is the value you should get that you use to update your parameters lambda. And again, the expectations are with respect to ZI drawn from the distributions QI. So the very last step of this derivation is we need to work out what these two expectations are. And so the very first term EZI transpose I guess is just mu of ZI give XI transpose because the QI distribution has mean given by [inaudible]. To work out the other term, let me just remind you that if you have a random variable Z that's Gaussian with mean mu and covariance sigma, then the covariance sigma is EZZ transpose minus EZ EZ transpose. That's one of the definitions of the covariance, and so this implies that EZZ transpose equals sigma plus EZ EZ transpose. And so this second term here becomes sigma ZI to the mu I plus – given XI. Okay?

And so that's how you compute E of ZI transpose and E of ZI ZI transpose and substitute them back into this formula. And you would then have your M step update to the parameter matrix lambda. And the last thing I want to point out in this derivation is that it turns out – it's probably because of the name EM algorithm, expectation maximization, one common mistake for the EM algorithm is in the E step, some want to take the expectation of the random variable Z, and then in the M step they just plug in the expected value everywhere you see it. So in particular, one common mistake in deriving EM for factor analysis is to look at this and say, "Oh, look. I see ZZ transpose. Let's just plug in the expected value under the Q distribution." And so plug in that – mu of ZI given XI times mu of ZI given XI transpose – into that expectation, and that would be an incorrect derivation of EM because it's missing this other term, sigma of ZI given XI. So one common misconception for EM is that in the E step you just compute the expected value of the hidden random variable, and the M step you plug in the expected value. It turns out in some algorithms that turns out to be the right thing to do. In the mixture of Gaussians and the mixture of [inaudible] models, that would actually give you the right answer, but in general the EM algorithm is more complicated than just taking the expected values of the random variables and then pretending that they were sort of observed at the expected values.

So I wanna go through this just to illustrate that step as well. So just to summarize the three key things to keep in – that came up in this variation were, 1.) That for the E step, we had a continuous Gaussian random variable, and so to compute the E step, we actually compute the mean and covariance of the distribution QI. The second thing that came up was in the M step when you see these integrals, sometimes if you interpret that as expectation then the rest of the math becomes much easier. And the final thing was again in the M step, the EM algorithm is derived by a certain maximization problem that we solve. It is not necessarily just plugging the expected value of ZI everywhere.

Let's see. I feel like I just did a ton of math and wrote down way too many equations. And even doing this, I was skipping many steps. So you can go to the lecture notes to see all the derivations of the steps I skipped, like how you actually take derivatives with respect to the matrix lambda, and how to compute the updates for the other parameters as well, for mu and for psi, because this is only for lambda. And so that's the factor analysis algorithm. Justin?

Student:I was just wondering in the step in the lower right board, you said that the second term doesn't have any parameters that we're interested in. The first term has all the parameters, so we'll [inaudible], but it seems to me that QI has a lot of parameters [inaudible].

Instructor (Andrew Ng):I see. Right. Let's see. So the question was doesn't the term QI have parameters? So in the EM algorithm QI is – it actually turns out in the EM algorithm, sometimes P of ZI may have parameters, but QI of ZI may never have any parameters. In the specific case of factor analysis, P of ZI doesn't have parameters. In other examples, the mixture of Gaussian models say, ZI was a multinomial random variable, and so in that example PI of ZI has parameters, but it turns out that Q of ZI will never have any parameters.

And in particular, QI of ZI is going to be – is this Gaussian distribution – is Gaussian with mean given by mu of ZI given XI and covariance sigma ZI given XI. And so it's true that mu and sigma may themselves have depended on the values of the parameters I had in the previous iteration of EM, but the way to think about Q is I'm going to take the parameters from the previous iteration of the algorithm and use that to compute what QI of ZI is. And that's the E second EM algorithm. And then once I've computed what QI of Z is, then this is a fixed distribution. I'm gonna use these fixed values for mu and sigma, and just keep these two values fixed as I run the M step.

Student: So that's – I guess I was confused because in the second point over there's a lot of – it looks like they're parameters, but I guess they're old iterations of the parameters.

Instructor (Andrew Ng): Oh, yeah. Yes, you're right. When I wrote down QI of ZI that was a function of – so yeah – the parameters from the previous iteration. And I want to compute the new set of parameters. Okay. More questions? So this is probably the most math I'll ever do in a lecture in this entire course. Let's now talk about a different algorithm. Actually, which board was I on? So what I want to do now is talk about an algorithm called principal components analysis, which is often abbreviated PCA. Here's the idea. PCA has a very similar idea as factor analysis, but it sort of maybe gets to the problem a little more directly than just factor analysis.

So the question is given – so we're still doing unsupervised learning, so given a training set of M examples where each XI is an N-dimensional vector as usual, what I like to do is reduce it to a lower dimensional data set where K is strictly less than N, and quite often will be much smaller than N. So I'll give a couple examples of why we want to do this. Imagine that you're given a data set that contains measurements and unknown to you – measurements of, I don't know, people or something – and unknown to you, whoever collected this data actually included the height of the person in centimeters as well as the height of the person in inches. So because of rounding off to the nearest centimeter or rounding off to the nearest inch the values won't exactly match up, but along two dimensions of this data anyway, it'll lie extremely close to a straight line, but it won't lie exactly on a straight line because of rounding off to the nearest centimeter or inch, but lie very close to the straight line.

And so we have a data set like this. It seems that what you really care about is that axis, and this axis is really the variable of interest. That's maybe the closest thing you have to the true height of a person. And this other axis is just noise. So if you can reduce the dimension of this data from two-dimensional to one-dimensional, then maybe you can get rid of some of the noise in this data. Quite often, you won't know

that this was cm and this was inches. You may have a data set with a hundred attributes, but you just didn't happen to notice one was cm and one was inches. Another example that I sometimes think about is some of you know that my students and I work with [inaudible] helicopters a lot. So we imagined that you take surveys of quizzes, measurements of radio control helicopter pilots. Maybe one axis you have measurements of your pilot's skill, how skillful your helicopter pilot is, and on another axis maybe you measure how much they actually enjoy flying. And maybe – this is really – maybe this is actually roughly one-dimensional data, and there's some variable of interest, which I'll call maybe pilot attitude that somehow determines their skill and how much they enjoy flying. And so again, if you can reduce this data from two dimensions to one-dimensional, maybe you'd have a slightly better of measure of what I'm calling loosely pilot attitude, which may be what you really wanted to [inaudible]. So let's talk about an algorithm to do this, and I should come back and talk about more applications of PCA later.

So here's the algorithm. Before running PCA normally you will preprocess the data as follows. I'll just write this out, I guess. I know some of you are writing. So this is maybe just an unnecessary amount of writing to say that compute a mean of my training sets and subtract out the means, so now I've zeroed out the mean of my training sets. And the other step is I'll compute the variance of each of features after zeroing out the mean, and then I'll divide each feature by the standard deviation so that each of my features now has equal variance. So these are some standard preprocessing steps we'll often do for PCA.

I'll just mention that sometimes the second step is usually done only when your different features are on different scales. So if you're taking measurements of people, one feature may be the height, and another may be the weight, and another may be the strength, and another may be how they age or whatever, all of these quantities are on very different scales, so you'd normally normalize the variance. Sometimes if all the XIs are the same type of thing – so for example if you're working with images and the XIJs are the pixels that they're – are on the same scale because they're all pixel intensity values ranging from zero to 255, then you may omit this step. So after preprocessing, let's talk about how we would find the main axes along which the data varies. How would you find a principal axes of variations [inaudible]? So to do that let me describe – let me just go through one specific example, and then we'll formalize the algorithm. Here's my training set comprising five examples. It has roughly zero mean. And the variance under X1 and X2 axes are the same. There's X1, there's X2 axes. And so the principal axes and variation of this data is roughly the positive 45-degree axis, so what I'd like to do is have my algorithm conclude that this direction that I've drawn with this arrow U is the best direction onto which to project the data, so the axis by which my data really varies.

So let's think about how we formalize. One thing we want to look at is suppose I take that axis – this is really the axis onto which I want to project – that I want to use to capture most of the variation of my data. And then when I take my training set and project it onto this line, then I get that set of points. And so you notice that those dots, the projections of my training sets onto this axis, has very large variance. In contrast, if I were to choose a different direction – let's say this is really [inaudible] the worst direction onto which to project my data. If I project all my data onto this axis, then I find that the projects of my data onto the purple line, onto this other line has much smaller variance, that my points are clustered together much more tightly. So one way to formalize this notion of finding the main axis of variations of data is I would like to find the vector U, I would like to find the direction U so that when I project my data onto that direction, the projected points vary as much as possible. So in other words, I'm gonna find a direction so that when I project my data onto it, the projections are largely widely spaced out. There's large variance.

So let's see. So I want to find the direction U. Just as a reminder, if I have a vector U of norm one, then the length of our vector XI projected – then the vector XI projected onto U has length XI transpose U. To project a vector onto – to project X onto unit vector, the length of the projection's just XI transpose U. And so to formalize my PCA problem, I'm going to choose a vector U to maximize – so I choose U and this is subject to the constraint that the norm of U, that the length of U is one – but I'm going to maximize – let's see, sum from I equals one to M – the length of the projection of the vectors X onto U. In particular, I want the sum of square distances of the projections to be far from the origin. I want the projections of X onto U to have large variance.

And just to simplify some of the math later, let me put a one over M in front. And so that quantity on the right is equal to one over M. Let's see. U transpose XI times XI transpose U, and so can simplify and I

get − this is U transpose times [inaudible] U. So I want to maximize U transpose times some matrix times U subject to the constraint that the length of U must be equal to one. And so some of you will recognize that this means U must be the principal eigenvector of this matrix in the middle. So let me just write that down and say a few more words about it.

So this implies that U is the principal eigenvector of the matrix, and I'll just call this matrix sigma. It actually turns out to be a covariance matrix, [inaudible] equals one to M transpose. Actually, let's check. How many of you are familiar with eigenvectors? Oh, cool. Lots of you, almost all of you. Great. What I'm about to say is very extremely familiar, but it's worth saying anyway. So if you have a matrix A and a vector U, and they satisfy AU equals lambda U, then this is what it means for U to be an eigenvector of the matrix A. And the value lambda here is called an eigenvalue. And so the principal eigenvector is just the eigenvector that corresponds to the largest eigenvalue. One thing that some of you may have seen, but I'm not sure that you have, and I just wanna relate this to stuff that you already know as well, is that optimization problem is really maximize U transpose U subject to the norm of U is equal to one and − let me write that constraint as that U transpose U equals one.

And so to solve this constrained optimization problem, you write down the Lagrangian [inaudible] lambda, where that's the Lagrange multiplier because there's a constraint optimization. And so to actually solve this optimization, you take the derivative of L with respect to U and that gives you sigma U minus lambda U. You set the derivative equal to zero and this shows that sigma U equals lambda U, and therefore the value of U that solves this constraint optimization problem that we care about must be an eigenvector of sigma. And in particular it turns out to be the principal eigenvector. So just to summarize, what have we done? We've shown that given a training set, if you want to find the principal axis of a variation of data, you want to find the 1-D axis on which the data really varies the most, what we do is we construct the covariance matrix sigma, the matrix sigma that I wrote down just now, and then you would find the principal eigenvector of the matrix sigma. And this gives you the best 1-D subspace onto which to project the data.

And more generally, you would choose − if you wanna K-dimensional subspace onto which project your date, you would then choose U1 through UK to be the top K eigenvectors of sigma. And by top K eigenvectors, I just mean the eigenvectors corresponding to the K highest eigenvalues. I guess I showed this only for the case of a 1-D subspace, but this holds more generally. And now the eigenvectors U1 through UK represent − give you a new basis with which to represent your data.

Student:[Inaudible] diagonal?

Instructor (Andrew Ng): Let's see. So by convention, the PCA will choose orthogonal axes. So I think this is what I'm saying. Here's one more example. Imagine that you're a three-dimensional set, and imagine that your three-dimensional data set − it's very hard to draw in 3-D on the board, so just let me try this. Imagine that here my X1 and X2 axes lie on the plane of the board, and the X3 axis points directly out of the board. Imagine that you have a data set where most of the data lies on the plane of the board, but there's just a little bit of fuzz. So imagine that the X3 axis points orthogonally out of the board, and all the data lies roughly in the plane of the board, but there's just a tiny little bit of fuzz so that some of the data lies just a couple of millimeters off the board.

So you run a PCA on this data. You find that U1 and U2 will be some pair of bases that essentially lie in the plane of the board, and U3 will be an orthogonal axis at points roughly out of the plane of the board. So if I reduce this data to two dimensions, then the bases U1 and U2 now give me a new basis with which to construct my lower dimensional representation of the data.

Just to be complete about what that means, previously we have say fairly high dimensional input data XI and RN, and now if I want to represent my data in this new basis given by U1 up to UK, instead I would take each of my original training examples XI and I would now represent it or replace it with a different vector which I call YI, and that would be computed as U1 transpose XI U2 transpose XI. And so the YIs are going to be K-dimensional where K will be less − your choice of K will be less than N, so this represents a lower dimensional representation of your data, and serves as an approximate representation of your original data where you're using only K numbers to represent each training example rather than N numbers. Let's see.

Student:[Inaudible] to have eigenvectors [inaudible] trivial eigenvectors?

157

Instructor (Andrew Ng): Is it possible not to have eigenvectors but have trivial eigenvectors?

Student:[Inaudible] determined by [inaudible]? Is it a condition wherein [inaudible].

Instructor (Andrew Ng): Let's see. What do you mean by trivial eigenvectors?

Student:[Inaudible] linear algebra from [inaudible].

Instructor (Andrew Ng): Oh, okay. Yes, so I see. Let me see if I can get this right. So there are some matrices that are – I think the term is degenerate – that don't have a full set of eigenvectors. Sorry, deficient I think is what it's called. Is that right, Ziko? Yeah. So some matrices are deficient and actually don't have a full set of eigenvectors, like an N by N matrix that does not have N distinct eigenvectors, but that turns out not to be possible in this case because the covariance matrix sigma is symmetric, and symmetric matrices are never deficient, so the matrix sigma will always have a full set of eigenvectors.

It's possible that if you – there's one other issue which is repeated eigenvalues. So for example, it turns out that in this example if my covariance matrix looks like this, it turns out that the identity of the first two eigenvectors is ambiguous. And by that I mean you can choose this to be U1 and this to be U2 or I can just as well rotate these eigenvectors and choose that to be U1 and that to be U2, or even choose that to be U1 and that to be U2, and so on.

And so when you apply PCA, one thing to keep in mind is sometimes eigenvectors can rotate freely within their subspaces when you have repeated eigenvalues or close to repeated eigenvalues. And so the way to think about the vectors U is think of as a basis with which to represent your data, but the basis vectors can sometimes rotate freely, and so it's not always useful to look at the eigenvectors one at a time, and say this is my first eigenvector capturing whatever, the height of a person, or this is my second eigenvector, and it captures their skill at [inaudible] or whatever. That's a very dangerous thing to do when you do PCA. What is meaningful is the subspace spanned by the eigenvectors, but looking at the eigenvectors one at a time is sometimes a dangerous thing to do because they can often freely. Tiny numerical changes can cause eigenvectors to change a lot, but the subspace spanned by the top K eigenvectors will usually be about the same.

It actually turns out there are multiple possible interpretations of PCA. I'll just give one more without proof, which is that [inaudible] whatever – given a training set like this, another view of PCA is – and let me just choose a direction. This is not the principal components. I choose some direction and project my data onto it. This is clearly not the direction PCA would choose, but what you can think of PCA as doing is choose a subspace onto which to project your data so as to minimize the sum of squares differences between the projections and the [inaudible] points. So in other words, another way to think of PCA is trying to minimize the sum of squares of these distances between the dots, the points X and the dots onto which I'm projecting the data. It turns out they're actually – I don't know. There's sort of nine or ten different interpretations of PCA. This is another one. There are a bunch of ways to derive PCA. You get play with some PCA ideas more in the next problem set.

What I want to do next is talk about a few applications of PCA. Here are some ways that PCA is used. One is visualization. Very often you have high dimensional data sets. Someone gives you a 50-dimensional data set, and it's very hard for you to look at a data set that's 50-dimensional and understand what's going on because you can't plot something in 50 dimensions. So common practice if you want to visualize a very high dimensional data set is to take your data and project it into say a 2-D plot, or project it into a 3-D plot, so you can render like a 3-D display on a computer so you can better visualize the data and look for structure. One particular example that I learned about of doing this recently was in Krishna Shenoy's lab here in Stanford in which he had readings from 50 different parts of a monkey brain. I actually don't know it was the number 50. It was tens of different parts of the monkey brain, and so you'd have these 50-dimensional readings, 50-dimensional vectors correspond to different amounts of electrical activity in different parts of the monkey brain. It was actually 50 neurons, but tens of neurons, but it was tens of neurons in the monkey brain, and so you have a 50-dimensional time series, and it's very hard to visualize very high dimensional data.

But what he understood is that – was use PCA to project this 50-dimensional data down to three dimensions, and then you can visualize this data in three dimensions just using 3-D plot, so you could visualize what the monkey is thinking over time. Another common application of PCA is compression, so if

you have high dimensional data and you want to store it with [inaudible] numbers, clearly PCA is a great way to do this. It turns out also that sometimes in machine learning, sometimes you're just given extremely high dimensional input data, and for computational reasons you don't want to deal with such high dimensional data. And so fairly – one common use of PCA is taking very high dimensional data and represent it with low dimensional subspace – it's that YI is the representation I wrote down just now – so that you can work with much lower dimensional data. And it turns out to be it's this sort of – just seems to make a fact of life that when you're given extremely high dimensional data almost all the time just in practice – of all the high dimensional data sets I've ever seen, very high dimensional data sets often have all their points lying on much lower dimensional subspaces, so very often you can dramatically reduce the dimension of your data and really be throwing too much of the information away. So let's see. If you're learning algorithms, it takes a long time to run very high dimensional data. You can often use PCA to compress the data to lower dimensions so that your learning algorithm runs much faster. And you can often do this with sacrificing almost no performance in your learning algorithm.

There's one other use of PCA for learning which is – you remember when we talked about learning theory, we said that the more features you have, the more complex your hypothesis class, if say you're doing linear classification. If you have more features, you have lots of features, then you may be more prone to overfitting. One other thing you could do with PCA is just use that to reduce the dimension of your data so that you have fewer features and you may be slightly less prone to overfitting. This particular application of PCA to learning I should say, it sometimes works. It often works. I'll actually later show a couple of examples where sort of do this and it works. But this particular application of PCA, I find looking in the industry, it also seems to be a little bit overused, and in particular – actually, let me say more about that later. We'll come back to that later. There's a couple of other applications to talk about. One is outlier detection or anomaly detection. The idea is suppose I give you a data set. You may then run PCA to find roughly the subspace on which your data lies. And then if you want to find anomalies in future examples, you then just look at your future examples and see if the lie very far from your subspace.

This isn't a fantastic anomaly detection algorithm. It's not a very good – the idea is if I give you a data set, you may find a little dimensional subspace on which it lies, and then if you ever find a point that's far from your subspace, you can factor it as an anomaly. So this really isn't the best anomaly detection algorithm, but sometimes this is done. And the last application that I want to go into a little bit more in detail is matching or to find better distance calculations. So let me say what I mean by this. I'll go into much more detail on this last one. So here's the idea. Let's say you want to do face recognition [inaudible]. So let's say you want to do face recognition and you have 100 of 100 images. So a picture of a face is like whatever, some array of pixels, and the pixels have different grayscale values, and dependent on the different grayscale values, you get different pictures of people's faces. And so [inaudible] you have 100 of 100 pixel images, and you think of each face as a 10,000-dimensional vector, which is very high dimensional. [Inaudible] cartoon to keep in mind, and you think of here's my plot of my 100-dimensional space, and if you look at a lot of different pictures of faces, each face will be along what will be some point in this 100,000-dimensional space. And in this cartoon, I want you to think of it as – I mean most of this data lies on a relatively low dimensional subspace because in this 10,000-dimensional space, really not all points correspond to valid images of faces. The vast majority of values, the vast majority of 10,000-dimensional images just correspond to random noise like looking things and don't correspond to a valid image of a face. But instead the space of possible face of images probably spans a much lower dimensional subspace. [Crosstalk]

Instructor (Andrew Ng): And so what we'll do is we'll use a low dimensional subspace on which the data lies, and in practice a PCA dimension of 50 would be fairly typical. So when you think of – there's some axes that really measure the shape or the appearance of the face, and there are some other axes that we're not interested in that are maybe just random noise.

So what we'll do is for face recognition I might give you a picture of a face and ask you what face looks the most similar to this. I'll give you a picture of someone and ask you can you find other pictures of the same person. And so the key step to do that is to look at two faces and to compare how similar these two faces are. So here's how we'll use PCA. Given a face there and a different face there, the way I'll measure the difference between these two faces is I won't just measure the Euclidian distance similarity. Instead, I'll take

the space and project it onto my 50-dimensional subspace – take this and project it onto my 50-dimensional subspace and measure the similarity between those two points on the subspace. And so when I do that, I may end up with a face here and a face here that look very far upon the original space, but when I project them onto the subspace, they may end up looking much more similar.

So what I want to show you a second on the laptop is given each of these training examples is a 10,000-dimensional vector, I can plot that as a grayscale image and I'd get like a picture of some person's face. What I'll also show you on the laptop will be plots of my eigenvectors. And so the eigenvectors also live in a 10,000-dimensional subspace. And I plot that.

You get some image that's – and these images are called eigenfaces. And what PCA is doing is essentially using linear combinations of these eigenface images to approximate these XIs, and it's the eigenfaces that span – so that these bases U1, U2 and so on that span hopefully the subspace along which my faces live. [Crosstalk]

Instructor (Andrew Ng): So here's a training set comprising aligned images like these, a much larger set than is shown here, and so when you run PCA, these are some of the plots of the UIs. Remember when I said plot the eigenvectors UI in the same way that it's plotting the training examples, and so you end up with the eigenvectors being grayscale images like these, and what approximate images of faces with linear combinations of these. So I said it's dangerous to look at individual eigenvectors and you really shouldn't do that, but let me just do that a little bit anyway. So if you look at the first eigenvector, this corresponds roughly to whether the face is illuminated from the left or the right. So depending on how heavy the weight is on this image, that roughly captures variation in illumination. The second image is hard to tell. It seems to be capturing variation in overall brightness of the face. The third eigenvector capturing roughly how much of a shadow or maybe a beard or something a person has and so forth. It's dangerous to look at individual eigenvectors, but it sort of slightly informs the look of it.

Here's an example of a specific application of eigenfaces. This is from Sandy Pentland – Alex Pentland's lab at MIT. In this display, this upper leftmost image is the input to the eigenface's algorithm. The algorithm is then asked to go through a database and find the most similar faces. This second image here is what it thinks is the most similar face of the input. The next one over is the second most similar. The next one over is the third most similar and so on. And so using eigenfaces, and by measuring differences between faces in that lower dimensional subspace, it is able to do a reasonable job identifying pictures of the same person even with faces of the person removed. And the next row shows the next most similar faces and so on, so this one is the fourth most similar face and so on. This is a usual application of eigenfaces.

The last thing I wanna say is just when people tell me about machine learning problems, I do often recommend they try PCA for different things, so PCA is a useful thing to know to use for compression, visualization, and so on. But in industry, I just tend to see PCA used slightly more often. There are also some times where you see people using PCA when they really shouldn't. So just one final piece of advice for use of PCA is before you use it, also think about whether you could just do it with the original training data XI without compressing it, since I've also definitely seen people compress the data when they really didn't need to. But having said that, I also often advise people to use PCA for various problems and it often works. Okay. Sorry about running late. So let's wrap up for today.

[End of Audio]
Duration: 82 minutes

1.15 Machine Learning Lecture 15[15]

Figure 1.15

MachineLearning-Lecture15

Instructor (Andrew Ng): All right. Good morning. Welcome back. So a couple quick announcements before I go into today's technical material. So unfortunately, a long time ago, I agreed to give a few [inaudible] in Japan, so I won't be here on Wednesday. So the next lecture will be, I shall be taping it ahead of time. In particular, instead of a lecture on Wednesday, I'll be giving Wednesday's lecture tomorrow instead at 1:30 p.m. in the Skeeling 191 auditorium.

So those here, you come to class at the usual time, the taped lecture will be taped ahead of time, but then shown in this room at this time. But if any of you are free tomorrow at 1:30 p.m., I should be grateful if you can come to that, alternative of the taping of the lecture, which will be at 1:30 p.m. Skeeling 191.

I'll also send an email out with the details, so you don't need to write it down. But if any of you can make it to that, it'll make it much easier and more fun for me to teach to a live audience rather than to teach on tape to an empty room. So I'll send an email about that. We'll do this for this Wednesday's lecture as well as for the lecture on the Wednesday after the Thanksgiving break.

In case you don't want to deal with it, and this is the only time you make it, you can just show up at the usual time, and the lecture will be presented here in this room as well, and online, of course. Let's see. For the same reason, my next two office hours, which are usually on Fridays will also be moved to the following Monday because I'll be traveling. So if you're planning to come to my office hours either this week or the week after Thanksgiving, just take a look at the course website for the revised time.

In case you want to meet with me, but can't make the new time on the course website, you can also send me an email. We'll try to set up a time outside that. Okay?

But again, if you're free at 1:30 p.m. tomorrow, either those watching this online now or people in this classroom, I would be grateful if you can come to Skeeling 191. If there are any of you that couldn't ever make this regular class time and so always watch it online, you can come see a live lecture, for once, those of you watching online.

Are there any administrative questions about that? Okay? Let's go into today's technical material.

Welcome back. What I want to do today is continue a discussion of principal components analysis, or PCA. In particular, there's one more application that I didn't get to in the last lecture on [inaudible] indexing, LSI. Then I want to spend just a little time talking about how to implement PCA, especially for very large problems. In particular, I'll spend just a little bit of time talking about singular value decomposition, or the SVD implementation of principal component analysis.

So the second half of today's lecture, I want to talk about the different algorithm called independent component analysis, which is, in some ways, related to PCA, but in many other ways, it also manages to accomplish very different things than PCA. So with this lecture, this will actually wrap up our discussion on unsupervised learning. The next lecture, we'll start to talk about reinforcement learning algorithms.

Just to recap where we were with PCA, principal component analysis, I said that in PCA, we imagine that we have some very high dimensional data that perhaps lies approximately on some low dimensional subspace. So if you had the data set like this, you might find that that's the first principal component of the data, and that's the second component of this 2-D data.

To summarize the algorithm, we have three steps. The first step of PCA was to normalize the data to zero mean and [inaudible]. So tracked out the means of your training examples. So it now has zero means, and then normalize each of your features so that the variance of each feature is now one.

---
[15]This content is available online at <http://cnx.org/content/m45930/1.3/>.

The next step was [inaudible] computical variance matrix of your zero mean data. So you compute it as follows. The sum of all the products, and then you find the top K eigen vectors of sigma. So last time we saw the applications of this. For example, one of the applications was to eigen faces where each of your training examples, XI, is an image. So if you have 100 by 100 images, if your pictures of faces are 100 pixels by 100 pixels, then each of your training examples, XI, will be a 10,000 dimensional vector, corresponding to the 10,000 grayscale intensity pixel values.

There are 10,000 pixel values in each of your 100 by 100 images. So the eigen faces application was where the training example comprised pictures of faces of people. Then we ran PCA, and then to measure the distance between say a face here and a face there, we would project both of the face images onto the subspace and then measure the distance along the subspace. So in eigen faces, you use something like 50 principle components.

So the difficulty of working with problems like these is that in step two of the algorithm, we construct the covariance matrix sigma. The covariance matrix now becomes a 10,000 by 10,000 dimensional matrix, which is huge. That has 100 million entries, which is huge.

So let's apply PCA to very, very high dimensional data, used as a point of reducing the dimension. But step two of this algorithm had this step where you were constructing [inaudible]. So this extremely large matrix, which you can't do. Come back to this in a second.

It turns out one of the other frequently-used applications of PCA is actually to text data. So here's what I mean. Remember our vectorial representation of emails? So this is way back when we were talking about supervised learning algorithms for a stand classification. You remember I said that given a piece of email or given a piece of text document, you can represent it using a very high-dimensional vector by taking — writing down a list of all the words in your dictionary. Somewhere you had the word learn, somewhere you have the word study and so on.

Depending on whether each word appears or does not appear in your text document, you put either a one or a zero there. This is a representation we use on an electrode five or electrode six for representing text documents for when we're building [inaudible] based classifiers for [inaudible].

So it turns out one of the common applications of PCA is actually this text data representations as well. When you apply PCA to this sort of data, the resulting algorithm, it often just goes by a different name, just latent semantic indexing. For the sake of completeness, I should say that in LSI, you usually skip the preprocessing step.

For various reasons, in LSI, you usually don't normalize the mean of the data to one, and you usually don't normalize the variance of the features to one. These are relatively minor differences, it turns out, so it does something very similar to PCA.

Normalizing the variance to one for text data would actually be a bad idea because all the words are — because that would have the affect of dramatically scaling up the weight of rarely occurring words. So for example, the word aardvark hardly ever appears in any document. So to normalize the variance of the second feature to one, you end up — you're scaling up the weight of the word aardvark dramatically. I don't understand why [inaudible].

So let's see. [Inaudible] the language, something that we want to do quite often is, give it two documents, XI and XJ, to measure how similar they are. So for example, I may give you a document and ask you to find me more documents like this one. We're reading some article about some user event of today and want to find out what other news articles there are. So I give you a document and ask you to look at all the other documents you have in this large set of documents and find the documents similar to this.

So this is typical text application, so to measure the similarity between two documents in XI and XJ, [inaudible] each of these documents is represented as one of these high-dimensional vectors. One common way to do this is to view each of your documents as some sort of very high-dimensional vector. So these are vectors in the very high-dimensional space where the dimension of the vector is equal to the number of words in your dictionary.

So maybe each of these documents lives in some 50,000-dimension space, if you have 50,000 words in your dictionary. So one nature of the similarity between these two documents that's often used is what's the angle between these two documents. In particular, if the angle between these two vectors is small, then the

two documents, we'll consider them to be similar. If the angle between these two vectors is large, then we consider the documents to be dissimilar.

So more formally, one commonly used heuristic, the national language of processing, is to say that the similarity between the two documents is a co-sine of the angle theta between them. For similar values, anyway, the co-sine is a decreasing function of theta. So the smaller the angle between them, the larger the similarity. The co-sine between two vectors is, of course, just [inaudible] divided by – okay? That's just the linear algebra or the standard geometry definition of the co-sine between two vectors.

Here's the intuition behind what LSI is doing. The hope, as usual, is that there may be some interesting axis of variations in the data, and there maybe some other axis that are just noise. So by projecting all of your data on lower-dimensional subspace, the hope is that by running PCA on your text data this way, you can remove some of the noise in the data and get better measures of the similarity between pairs of documents.

Let's just delve a little deeper into those examples to convey more intuition about what LSI is doing. So look further in the definition of the co-sine similarity measure. So the numerator or the similarity between the two documents was this inner product, which is therefore sum over K, XIK, XJK. So this inner product would be equal to zero if the two documents have no words in common. So this is really – sum over K – indicator of whether documents, I and J, both contain the word, K, because I guess XIK indicates whether document I contains the word K, and XJK indicates whether document J contains the word, K.

So the product would be one only if the word K appears in both documents. Therefore, the similarity between these two documents would be zero if the two documents have no words in common. For example, suppose your document, XI, has the word study and the word XJ, has the word learn. Then these two documents may be considered entirely dissimilar.

[Inaudible] effective study strategies. Sometimes you read a news article about that. So you ask, what other documents are similar to this? If there are a bunch of other documents about good methods to learn, than there are words in common. So similarity [inaudible] is zero.

So here's a cartoon of what we hope [inaudible] PCA will do, which is suppose that on the horizontal axis, I plot the word learn, and on the vertical access, I plot the word study. So the values take on either the value of zero or one. So if a document contains the words learn but not study, then it'll plot that document there. If a document contains neither the word study nor learn, then it'll plot that at zero, zero.

So here's a cartoon behind what PCA is doing, which is we identify lower dimensional subspace. That would be sum – eigen vector, we get out of PCAs. Now, supposed we have a document about learning. We have a document about studying. The document about learning points to the right. Document about studying points up. So the inner product, or the co-sine angle between these two documents would be – excuse me. The inner product between these two documents will be zero. So these two documents are entirely unrelated, which is not what we want.

Documents about study, documents about learning, they are related. But we take these two documents, and we project them onto this subspace. Then these two documents now become much closer together, and the algorithm will recognize that when you say the inner product between these two documents, you actually end up with a positive number. So LSI enables our algorithm to recognize that these two documents have some positive similarity between them.

So that's just intuition about what PCA may be doing to text data. The same thing goes to other examples and the words study and learn. So you have – you find a document about politicians and a document with the names of prominent politicians. That will also bring the documents closer together, or just any related topics, they end up [inaudible] points closer together and just lower dimensional space.

Question about this? Interviewee:

[Inaudible].

Instructor (Andrew Ng): Which ones? This one? Interviewee:

No, the line.

Instructor (Andrew Ng): Oh, this one. Oh, yes. Thank you. [Inaudible]. So let's talk about how to actually implement this now. Okay. How many of you know what an SVD or single value decomposition is? Wow, that's a lot of you. That's a lot more than I thought. Curious. Did you guys learn it as under grads

or as graduate students?

All right. Let me talk about it anyway. I wasn't expecting so many of you to know what SVD is, but I want to get this on tape, just so everyone else can learn about this, too. So I'll say a little bit about how to implement PCA. The problem I was eluding to just now was that when you have these very high-dimensional vectors, than sigma is a large matrix.

In particular, for our text example, if the vectors XI are 50,000 dimensional, then the covariance matrix will be 50,000 dimensional by 50,000 dimensional, which is much too big to represent explicitly. I guess many of you already know this, but I'll just say it anyway. It turns out there's another way to implement PCA, which is if A is any N by N matrix, than one of the most remarkable results of linear algebra is that the matrix, A, can be decomposed into a singular value decomposition.

What that means is that the matrix, A, which is N by N, can always be decomposed into a product of three matrixes. U is N by N, D is a square matrix, which is N by N, and V is also N by N. D is going to be diagonal. Zeros are on the off-diagonals, and the values sigma I are called the singular values of the matrix A.

Almost all of you said you learned this as a graduate student, rather than as an under grad, so it turns out that when you take a class in undergraduate linear algebra, usually you learn a bunch of decomposition. So you usually learn about the QLD composition, maybe the LU factorization of the matrixes. Most under grad courses don't get to talk about singular value decompositions, but at least in − almost everything I do in machine learning, you actually find that you end up using SVDs much more than any of the decompositions you learned in typical under grad linear algebra class.

So personally, I [inaudible] an SVD dozens of times in the last year, but LU and QRD compositions, I think I used the QRD composition once and an LU decomposition in the last year. So let's see. I'll say a bit more about this. So I'm going to draw the picture, I guess.

For example, if A is an N by N matrix, it can be decomposed into another matrix, U, which is also N by N. It's the same size, D, which is N by N. Another square matrix, V transpose, which is also N by N. Furthermore, in a singular value decomposition, the columns of the matrix, U, will be the eigen vectors of A transpose, and the columns of V will be the eigen vectors of A transpose A.

To compute it, you just use the SVD commands in Matlab or Octave. Today, say the art in numerical linear algebra is that SVD, singular value decompositions, and matrixes can be computed extremely [inaudible]. We've used a package like Matlab or Octave to compute, say, the eigen vectors of a matrix. So if SVD routines are even more numerically stable than eigen vector routines for finding eigen vector in the matrix. So you can safely use a routine like this, and similar to the way they use a square root command without thinking about how it's computed.

You can compute the square root of something and just not worry about it. You know the computer will give you the right answer. For most reasonably-sized matrixes, even up to thousands by thousands matrixes, the SVD routine, I think of it as a square root function. If you call it, it'll give you back the right answer. You don't have to worry too much about it.

If you have extremely large matrixes, like a million by a million matrixes, I might start to worry a bit, but a few thousand by a few thousand matrixes, this is implemented very well today. Interviewee: [Inaudible].

Instructor (Andrew Ng):What's the complexity of SVD? That's a good question. I actually don't know. I want to guess it's roughly on the order of N-cubed. I'm not sure. [Inaudible] algorithms, so I think − I don't know what's known about the conversion of these algorithms.

The example I drew out was for a facts matrix, and a matrix is [inaudible]. In the same way, you can also call SVD on the tall matrix, so it's taller than it's wide. It would decompose it into − okay? A product of three matrixes like that.

The nice thing about this is that we can use it to compute eigen vectors and PCA very efficiently. In particular, a covariance matrix sigma was this. It was the sum of all the products, so if you go back and recall the definition of the design matrix − I think I described this in lecture two when we derived the close form solution to these squares [inaudible] these squares. The design matrix was this matrix where I took my examples and stacked them in rows. They call this the design matrix [inaudible].

So if you construct the design matrix, then the covariance matrix sigma can be written just X transposing. That's X transposed, and [inaudible]. Okay? I hope you see why the X transpose X gives you the sum of products of vectors. If you aren't seeing this right now, just go home and convince yourself [inaudible] if it's true.

To get the top K eigen vectors of sigma, you would take sigma and decompose it using the – excuse me. You would take the matrix X, and you would compute as SVD. So you get USV transpose. Then the top three columns of U are the top K eigen vectors of X transpose X, which is therefore, the top K eigen vectors of your covariance matrix sigma.

So in our examples, the design matrix may be, say R. If you have 50,000 words in your dictionary, than the design matrix would be RM by 50,000. [Inaudible] say 100 by 50,000, if you have 100 examples. So X would be quite tractable to represent and compute the SVD, whereas the matrix sigma would be much harder to represent. This is 50,000 by 50,000. So this gives you an efficient way to implement PCA.

The reason I want to talk about this is in previous years, I didn't talk [inaudible]. The class projects, I found a number of students trying to implement SVD on huge problems and [inaudible], so this is a much better to implement PCA if you have extremely high dimensional data. If you have low dimensional data, if you have 50 or 100 dimensional data, then computing sigma's no problem. You can do it the old way, but otherwise, use the SVD to implement this.

Questions about this?

The last thing I want to say is that in practice, when you want to implement this, I want to say a note of caution. It turns out that for many applications of – let's see. When you apply SVD to these wide – yeah. Interviewee:

Just a quick question. Are the top K columns of U or V because X transposed X is V transpose, right?

Instructor (Andrew Ng):Let's see. Oh, yes. I think you're right. I think you're right. Let's see. Is it top K columns of U or top K of V? Yeah, I think you're right. Is that right? Something bothers me about that, but I think you're right. Interviewee:

[Inaudible], so then X transpose X should be VDD. X is UDV, so X transpose X would be –

Instructor (Andrew Ng):[Inaudible]. If anyone thinks about this and has another opinion, let me know, but I think you're right. I'll make sure I get the details and let you know. Everyone's still looking at that. Tom, can you figure out the right answer and let me know? Male Speaker:

That sounds right.

Instructor (Andrew Ng):Okay. Cool. Okay. So just one last note, a note of caution. It turns out that in this example, I was implementing SVD with a wide matrix. So the matrix X was N by N. It turns out when you find the SVD decomposition of this, it turns out that – let's see. Yeah, I think you're definitely right. So it turns out that we find the SVD of this, the right-most portion of this block of this matrix would be all zeros.

Also, when you compute the matrix, D, a large part of this matrix would be zeros. You have the matrix D transpose. So depending on what convention you use, for example, I think Matlab actually uses a convention of just cutting off the zero elements. So the Matlab uses the convention of chopping off the right-most half of the U matrix and chopping off the bottom portion of the D matrix.

I'm not sure if this even depends on the version of Matlab, but when you call SVD on Matlab or some other numerical algebra packages, there's slightly different conventions of how to define your SVD when the matrix is wider than it is tall. So just watch out for this and make sure you map whatever convention your numerical algebra library uses to the original computations.

It turns out if you turn Matlab [inaudible] or you're writing C code. There are many scientific libraries that can compute SVDs for you, but they're just slightly different in conventions for the dimensions of these matrixes. So just make sure you figure this out for the package that you use.

Finally, I just want to take the unsupervised learning algorithms we talked about and just put a little bit of broader context. This is partly in response to the questions I've gotten from students in office hours and elsewhere about when to use each of these algorithms. So I'm going to draw a two by two matrix. This is a little cartoon that I find useful.

One of the algorithms we talked about earlier, right before this, was factor analysis, which was – it was

– I hope you remember that picture I drew where I would have a bunch of point Z on the line. Then I had these ellipses that I drew. I hope you remember that picture. This was a factor analysis model which models the density effects [inaudible], right?

It was also a PCA, just now. So the difference between factor analysis and PCA, the way I think about it, is that factor analysis is a density estimation algorithm. It tries to model the density of the training example's X. Whereas PCA is not a probabilistic algorithm. In particular, it does not endow your training examples of any probabilistic distributions and directly goes to find the subspace.

So in terms of when to use factor analysis and when to use PCA, if your goal is to reduce the dimension of the data, if your goal is to find the subspace that the data lies on, then PCA directly tries to find the subspace. I think I would tend to use PCA.

Factor analysis, it sort of assumes the data lies on a subspace. Let me write a subspace here. So both of these algorithms sort of assume the data maybe lies close or on some low dimensional subspace, but fundamentally, factor analysis, I think of it as a density estimation algorithm. So that has some very high dimensional distribution. I want to model P of X, then the factor analysis is the algorithm I'm more inclined to use.

So even though you could in theory, I would tend to avoid trying to use factor analysis to identify a subspace the data set lies on. So [inaudible], if you want to do anomaly detection, if you want to model P of X so that if you have a very low probability of N, you can factor an anomaly, then I would tend to use factor analysis to do that density estimation.

So factor analysis and PCA are both algorithms that assume that your data lies in the subspace. The other cause of algorithms we talked about was algorithms that assumes the data lies in clumps or that the data has a few coherence to groups. So let me just fill in the rest of this picture. So if you think your data lies in clumps or lies in groups, and if it goes [inaudible] density estimation, then I would tend to use a mixture of [inaudible] algorithm.

But again, you don't necessarily want to endow your data of any probably semantics, so if you just want to find the clumps of the groups, then I'd be inclined to use a [inaudible] algorithm. So haven't seen anyone else draw this picture before, but I tend to organize these things this way in my brain. Hopefully this helps guide when you might use each of these algorithms as well, depending on whether you believe the data might lie in the subspace or whether it might bind in clumps or groups.

All right. That wraps up the discussion on PCA. What I want to do next is talk about independent component analysis, or ICA. Yeah. Interviewee:

I have a question about the upper right [inaudible]. So once you have all of the eigen vectors, [inaudible] how similar is feature I to feature J. You pick some eigen vector, and you take some dot products between the feature I and feature J and the eigen vector. But there's a lot of eigen vectors to choose from.

Instructor (Andrew Ng):Right. So Justin's question was having found my eigen vectors, how do I choose what eigen vector to use to measure distance. I'm going to start this up. So the answer is really – in this cartoon, I would avoid thinking about eigen vectors one other time. A better way to view this cartoon is that this is actually – if I decide to choose 100 eigen vectors, this is really 100 D subspace.

So I'm not actually projecting my data onto one eigen vector. This arrow, this cartoon, this denotes the 100-dimensional subspace [inaudible] by all my eigen vectors. So what I actually do is project my data onto the span, the linear span of eigen vectors. Then I measure distance or take inner products of the distance between the projections of the two points of the eigen vectors. Okay.

So let's talk about ICA, independent component analysis. So whereas PCA was an algorithm for finding what I call the main axis of variations of data, in ICA, we're going to try find the independent of components of variations in the data. So switch it to the laptop there, please.

We'll just take a second to motivate that. I'm going to do so by – although if you put on the – okay. This is actually a slide that I showed in lecture one of the cocktail party problem. Suppose you have two speakers at a cocktail party, and you have two microphones in the room, overlapping sets of two conversations. Then can you separate out the two original speaker sources? So I actually played this audio as well in the very first lecture, which is suppose microphone one records this. [Recording]

Instructor (Andrew Ng):So the question is, these are really two speakers, speaking independently of

each other. So each speaker is outputting a series of sound signals as independent of the other conversation going on in the room. So this being an supervised learning problem, the question is, can we take these two microphone recordings and feed it to an algorithm to find the independent components in this data? This is the output when we do so. [Recording]

Instructor (Andrew Ng):This is the other one. [Recording]

Instructor (Andrew Ng):Just for fun. [Inaudible]. These are audio clips I got from [inaudible]. Just for fun, let me play the other ones as well. This is overlapping microphone one. [Recording]

Instructor (Andrew Ng):Here's microphone two. [Recording]

Instructor (Andrew Ng):So given this as input, here's output one. [Recording]

Instructor (Andrew Ng):It's not perfect, but it's largely cleaned up the music. Here's number two. [Recording]

Instructor (Andrew Ng):Okay. Switch back to [inaudible], please. So what I want to do now is describe an algorithm that does that. Before I actually jump into the algorithm, I want to say two minutes of CDF, so cumulative distribution functions. I know most of you know what these are, but I'm just going to remind you of what they are.

Let's say you have a one-D random variable S. So suppose you have a random variable, S, and suppose it has a property density function [inaudible]. Then the CDF is defined as a function, or rather as F, which is the probability that the random variable, S, is less than the value given by that lower-case value, S. For example, if this is your [inaudible] density, than the density of the [inaudible] usually to note it lower-case phi. That's roughly a bell-shaped density.

Then the CDF or the Gaussian will look something like this. There'll be a capital function pi. So if I pick a value S like that, then the height of this – this is [inaudible] probability that my Gaussian random variable is less than that value there. In other words, the height of the function at that point is less than the area of the Gaussian density, up to the point S. As you move further and further to the right, this function will approach one, as you integrate more and more of this area of the Gaussian.

So another way to write F of S is the integral, the minus infinity to S of the density, DT. So something that'll come later is suppose I have a random variable, S, and I want to model the distribution of the random variable, S. So one thing I could do is I can specify what I think the density is. Or I can specify what the CDF is. These are related by this equation. F is the integral of P of S.

You can also recover the density by taking the CDF and taking the derivative. So F prime, take the derivative of the CDF, you get back the density. So this has come up in the middle of when I derive ICA, which is that there'll be a step where they need to assume a distribution for random variable, S. I can either specify the density for S directly, or I can specify the CDF. I choose to specify the CDF.

It has to be some function increasing from zero to one. So you can choose any function that looks like that, and in particular, pulling functions out of a hat that look like that. You can, for instance, choose a sigmoid function of CDF. That would be one way of specifying the distribution of the densities for the random variable S. So this will come up later.

Just [inaudible], just raise your hand if that is familiar to you, if you've seen that before. Great.

So let's start to derive our RCA, or our independent component analysis algorithm. Let's assume that the data comes from N original sources. So let's say there are N speakers in a cocktail party. So the original sources, I'm going to write as a vector, S as in RN. So just to be concrete about what I mean about that, I'm going to use SIJ to denote the signal from speaker J at time I.

Here's what I mean. So what is sound? When you hear sound waves, sound is created by a pattern of expansions and compressions in air. So the way you're hearing my voice is my mouth is causing certain changes in the air pressure, and then your ear is hearing my voice as detecting those changes in air pressure. So what a microphone records, what my mouth is generating, is a pattern. I'm going to draw a cartoon, I guess. Changes in air pressure. So this is what sound is.

You look at a microphone recording, you see these roughly periodic signals that comprise of changes in air pressure over time as the air pressure goes above and below some baseline air pressure. So this is what the speech signal looks like, say. So this is speaker one. Then what I'm saying is that – this is some time, T.

What I'm saying is that the value of that point, I'm going to denote as S, super script T, sub script one.

Similarly, speaker two, it's outputting some sound wave. Speaker voice will play that. It'll actually sound like a single tone, I guess. So in the same way, at the same time, T, the value of the air pressure generated by speaker two, I'll denote as ST 2.

So we observe XI equals A times SI, where these XIs are vectors in RN. So I'm going to assume that I have N microphones, and each of my microphones records some linear combination of what the speakers are saying. So each microphone records some overlapping combination of what the speakers are saying.

For example, XIJ, which is – this is what microphone J records at time, I. So by definition of the matrix multiplication, this is sum of AIKSJ. Oh, excuse me. Okay? So what my J – sorry. So what my J microphone is recording is some linear combination of all of the speakers. So at time I, what microphone J is recording is some linear combination of what all the speakers are saying at time I. So K here indexes over the N speakers.

So our goal is to find the matrix, W, equals A inverse, and just defining W that way. So we can recover the original sources as a linear combination of our microphone recordings, XI. Just as a point of notation, I'm going to write the matrix W this way. I'm going to use lower case W subscript one, subscript two and so on to denote the roles of this matrix, W.

Let's see. So let's look at why IC is possible. Given these overlapping voices, let's think briefly why it might be possible to recover the original sources. So for the next example, I want to say – let's say that each of my speakers outputs – this will sound like white noise. Can I switch the laptop display, please? For this example, let's say that each of my speakers outputs uniform white noise. So if that's the case, these are my axis, S1 and S2. This is what my two speakers would be uttering.

The parts of what they're uttering will look like a line in a square box if the two speakers are independently outputting uniform minus one random variables. So this is part of S1 and S2, my original sources. This would be a typical sample of what my microphones record. Here, at the axis, are X1 and X2. So these are images I got from [inaudible] on ICA. Given a picture like this, you can sort of look at this box, and you can sort of tell what the axis of this parallelogram are. You can figure out what linear transformation would transform the parallelogram back to a box.

So it turns out there are some inherent ambiguities in ICA. I'll just say what they are. One is that you can't recover the original indexing of the sources. In particular, if I generated the data for speaker one and speaker two, you can run ICA, and then you may end up with the order of the speakers reversed. What that corresponds to is if you take this picture and you flip this picture along a 45-degree axis. You take a 45-degree axis and reflect this picture across the 45-degree axis, you'll still get a box.

So there's no way for the algorithms to tell which was speaker No. 1 and which was speaker No. 2. The numbering or the ordering of the speakers is ambiguous. The other source of ambiguity, and these are the only ambiguities in this example, is the sign of the sources. So given my speakers' recordings, you can't tell whether you got a positive SI or whether you got back a negative SI.

In this picture, what that corresponds to is if you take this picture, and you reflect it along the vertical axis, if you reflect it along the horizontal axis, you still get a box. You still get back [inaudible] speakers. So it turns out that in this example, you can't guarantee that you've recovered positive SI rather than negative SI.

So it turns out that these are the only two ambiguities in this example. What is the permutation of the speakers, and the other is the sign of the speakers. Permutation of the speakers, there's not much you can do about that. It turns out that if you take the audio source and if you flip the sign, and you take negative S, and if you play that through a microphone it'll sound indistinguishable. So for many of the applications we care about, the sign as well as the permutation is ambiguous, but you don't really care about it.

Let's switch back to chalk board, please. It turns out, and I don't want to spend too much time on this, but I do want to say it briefly. It turns out the reason why those are the only sources of ambiguity – so the ambiguities were the permutation of the speakers and the signs. It turns out that the reason these were the only ambiguities was because the SIJs were non-Gaussian. I don't want to spend too much time on this, but I'll say it briefly.

Suppose my original sources, S1 and S2, were Gaussian. So suppose SI is Gaussian, would mean zero and identity covariance. That just means that each of my speakers outputs a Gaussian random variable. Here's a typical example of Gaussian data.

You will recall the contours of a Gaussian distribution with identity covariants looks like this, right? The Gaussian is a spherically symmetric distribution. So if my speakers were outputting Gaussian random variables, than if I observe a linear combination of this, there's actually no way to recover the original distribution because there's no way for me to tell if the axis are at this angle or if they're at that angle and so on. The Gaussian is a rotationally symmetric distribution, so I would no be able to recover the orientation in the rotation of this.

So I don't want to prove this too much. I don't want to spend too much time dwelling on this, but it turns out if your source is a Gaussian, then it's actually impossible to do ICA. ICA relies critically on your data being non-Gaussian because if the data were Gaussian, then the rotation of the data would be ambiguous. So regardless of how much data you have, even if you had infinitely large amounts of data, you would not be able to recover the matrix A or W.

Let's go ahead and divide the algorithm. To do this, I need just one more result, and then the derivation will be three lines. [Inaudible] many variables as N, which is the joint vector of the sound that all of my speakers that are emitting at any time. So let's say the density of S is P subscript S, capital S. So my microphone recording records S equals AS, equals W inverse S. Equivalently, S equals W sign of X.

So let's think about what is the density of X. So I have P of S. I know the density of S, and X is a linear combination of the S's. So let's figure out what is the density of X. One thing we could do is figure out what S is. So this is just − apply the density of S to W of S. So let's see. This is the probability of S, so we just figure out what S is.

S is W times X, so the probability of S is W times X, so the probability of X must be [inaudible]. So this is wrong. It turns out you can do this for probably mass functions but not for continuous density. So in particular, it's not correct to say that the probability of X is − well, you just figure out what S is.

Then you say the probability of S is applied to that. This is wrong. You can't do this with densities. You can't say the probability of S is that because it's a property density function. In particular, the right formula is the density of S applied to W times X, times the determinant of the matrix, W. Let me just illustrate that with an example.

Let's say the density for S is that. In this example, S is uniform over the unit interval. So the density for S looks like that. It's just density for the uniform distribution of zero one. So let me let X be equal to two times S. So this means A equals two. W equals one half. So if S is a uniform distribution over zero, one, then X, which is two times that, will be the uniform distribution over the range from zero to two.

So the density for X will be − that's one, that's two, that's one half, and that's one. Okay? Density for X will be indicator zero [inaudible] for X [inaudible] two times W, times one half.

So does that make sense? [Inaudible] computer density for X because X is now spread out across a wider range. The density of X is now smaller, and therefore, the density of X has this one half term here. Okay?

This is an illustration for the case of one-dimensional random variables, or S and X of one D. I'm not going to show it, but the generalization of this to vector value random variables is that the density of X is given by this times the determinant of the matrix, W. Over here, I showed the one dimensional [inaudible] generalization.

So we're nearly there. Here's how I can implement ICA. So my distribution on S, so I'm going to assume that my density on S is given by this as a product over the N speakers of the density − the product of speaker I emitting a certain sound. This is a product of densities. This is a product of distributions because I'm going to assume that the speakers are having independent conversations.

So the SI's independent for different values of I. So by the formula we just worked out, the density for X would be equal to that. I'll just remind you, W was A inverse. It was this matrix I defined previously so that SI equals WI [inaudible] X. So that's what's in there.

To complete my formulation for this model, the final thing I need to do is choose a density for what I think each speaker is saying. I need to assume some density over the sounds emitted by an individual speaker. So following the discussion I had right when the [inaudible] ICA, one thing I could do is I could choose the density for S, or equivalently, I could choose the CDF, the cumulative distribution function for S.

In this case, I'm going to choose a CDF, probably for historical reasons and probably for convenience. I need to choose the CDF for S, so what that means is I just need to choose some function that increases

from zero to what. I know I can't choose a Gaussian because we know you can't do ICA on Gaussian data. So I need some function increasing from zero to one that is not the cumulative distribution function for a Gaussian distribution.

So what other functions do I know that increase from zero to one? I just choose the CDF to be the sigmoid function. This is a commonly-made choice that is made for convenience. There is actually no great reason for why you choose a sigmoid function. It's just a convenient function that we all know and are familiar with that happens to increase from zero to one.

When you take the derivative of the sigmoid, and that will give you back your density. This is just not Gaussian. This is the main virtue of choosing the sigmoid. So there's really no rational for the choice of sigma. Lots of other things will work fine, too. It's just a common, reasonable default.

It turns out that one reason the sigma works well for a lot of data sources is that if this is the Gaussian. If you actually take the sigmoid and you take its derivative, you find that the sigmoid has [inaudible] than the Gaussian. By this I mean the density of the sigmoid dies down to zero much more slowly than the Gaussian. The magnitudes of the tails dies down as E to the minus S squared. For the sigmoid, the tails look like E to the minus S. So the tails die down as E to the minus S, around E to the minus S squared.

It turns out that most distributions of this property with [inaudible] tails, where the distribution decays to zero relatively slowly compared to Gaussian will work fine for your data. Actually, one other choice you can sometimes us is what's called the Laplacian distribution, which is that. This will work fine, too, for many data sources.

Sticking with the sigmoid for now, I'll just write down the algorithm in two steps. So given my training set, and as you show, this is an unlabeled training set, I can write down the log likelihood of my parameters. So that's − assembled my training examples, log of − times that. So that's my log likelihood. To learn the parameters, W, of this model, I can use the [inaudible] assent, which is just that. It turns out, if you work through the math, let's see. If P of S is equal to the derivative of the sigmoid, then if you just work through the math to compute the [inaudible] there. You've all done this a lot of times. I won't bother to show the details. You find that is equal to this.

Okay? That's just − you can work those out yourself. It's just math to compute the derivative of this with respect to W. So to summarize, given the training set, here's my [inaudible] update rule. So you run the [inaudible] to learn the parameters W. After you're done, you then output SI equals WXI, and you've separated your sources of your data back out into the original independent sources.

Hopefully up to only a permutation and a plus/minus sign ambiguity. Okay? So just switch back to the laptop, please? So we'll just wrap up with a couple of examples of applications of ICA.

This is actually a picture of our TA, Katie. So one of the applications of ICA is to process various types of [inaudible] recording data, so [inaudible]. This is a picture of a EEG cap, in which there are a number of electrodes you place on the − in this case, on Katie's brain, on Katie's scalp. So where each electrode measures changes in voltage over time on the scalp. On the right, it's a typical example of [inaudible] data where each electrode measures − just changes in voltage over time.

So the horizontal axis is time, and the vertical axis is voltage. So here's the same thing, blown up a little bit. You notice there are artifacts in this data. Where the circle is, where the data is circled, all the electrodes seem to measure in these very synchronized recordings. It turns out that we look at [inaudible] data as well as a number of other types of data, there are artifacts from heartbeats and from human eye blinks and so on.

So the cartoonist, if you imagine, placing the electrodes, or microphones, on my scalp, then each microphone is recording some overlapping combination of all the things happening in my brain or in my body. My brain has a number of different processes going on. My body's [inaudible] going on, and each electrode measures a sum of the different voices in my brain. That didn't quite come out the way I wanted it to.

So we can just take this data and run ICA on it and find out one of the independent components, what the independent process are going on in my brain. This is an example of running ICA. So you find that a small number of components, like those shown up there, they correspond to heartbeat, where the arrows − so those are very periodic signals. They come on occasionally and correspond to [inaudible] components of heartbeat.

You also find things like an eye blink component, corresponding to a sigmoid generated when you blink your eyes. By doing this, you can then subtract out the heartbeat and the eye blink artifacts from the data, and now you get much cleaner ICA data – get much cleaner EEG readings. You can do further scientific studies.

So this is a pretty commonly used preprocessing step that is a common application of ICA. [Inaudible] example is the application, again, from [inaudible]. As a result of running ICA on natural small image patches. Suppose I take natural images and run ICA on the data and ask what are the independent components of data. It turns out that these are the bases you get. So this is a plot of the sources you get.

This algorithm is saying that a natural image patch shown on the left is often expressed as a sum, or a linear combination, of independent sources of things that make up images. So this model's natural images is generated by independent objects that generate different ages in the image.

One of the fascinating things about this is that, similar to neuroscience, this has also been hypothesized as a method for how the human brain processes image data. It turns out, this is similar, in many ways, to computations happening in early visual processing in the human brain, in the mammalian brain. It's just interesting to see ages are the independent components of images.

Are there quick questions, because I'm running late. Quick questions before I close? Interviewee: [Inaudible] square matrix?

Instructor (Andrew Ng):Oh, yes. For the algorithms I describe, I assume A is a square matrix. It turns out if you have more microphones than speakers, you can also apply very similar algorithms. If you have fewer microphones than speakers, there's sort of an open research problem. The odds are that if you have one male and one female speaker, but one microphone, you can sometimes sort of separate them because one is high, one is low.

If you have two male speakers or two female speakers, then it's beyond the state of the art now to separate them with one microphone. It's a great research program.

Okay. Sorry about running late again. Let's close now, and we'll continue reinforcement learning next [inaudible].

[End of Audio]

Duration: 80 minutes

1.16 Machine Learning Lecture 16[16]

Figure 1.16

MachineLearning-Lecture16

Instructor (Andrew Ng):Okay, let's see. Just some quick announcements. For those of you taking 221, in 221 and 229, I said that in supervised learning there was about one lecture that would overlap and everything else was much more advanced in 229. And some of the reinforcement or anything else in 221, there's about one vector of overlap between 221 and 229, but then right after that, we actually go much further in 229 than we did in 221.

All right, so welcome back. What I want to do today is start a new chapter, a new discussion on machine learning and in particular, I want to talk about a different type of learning problem called reinforcement learning, so that's Markov Decision Processes, value functions, value iteration, and policy iteration. Both of these last two items are algorithms for solving reinforcement learning problems.

---

[16]This content is available online at <http://cnx.org/content/m45932/1.3/>.

As you can see, we're also taping a different room today, so the background is a bit different.

So just to put this in context, the first of the four major topics we had in this class was supervised learning and in supervised learning, we had the training set in which we were given sort of the "right" answer of every training example and it was then just a drop of the learning algorithms to replicate more of the right answers.

And then that was learning theory and then we talked about unsupervised learning, and in unsupervised learning, we had just a bunch of unlabeled data, just the x's, and it was the job in the learning algorithm to discover so-called structure in the data and several algorithms like cluster analysis, K-means, a mixture of all the sort of the PCA, ICA, and so on.

Today, I want to talk about a different class of learning algorithms that's sort of in between supervised and unsupervised, so there will be a class of problems where there's a level of supervision that's also much less supervision than what we saw in supervised learning. And this is a problem in formalism called reinforcement learning. So next up here are slides. Let me show you. As a moving example, here's an example of the sorts of things we do with reinforcement learning.

So here's a picture of – some of this I talked about in Lecture 1 as well, but here's a picture of the – we have an autonomous helicopter we have at Stanford. So how would you write a program to make a helicopter like this fly by itself? I'll show you a fun video. This is actually, I think, the same video that I showed in class in the first lecture, but here's a video taken in the football field at Stanford of using machine learning algorithm to fly the helicopter. So let's just play the video.

You can zoom in the camera and see the trees in the sky. So in terms of autonomous helicopter flight, this is written then by some of my students and me. In terms of autonomous helicopter flight, this is one of the most difficult aerobatic maneuvers flown and it's actually very hard to write a program to make a helicopter do this and the way this was done was with what's called a reinforcement learning algorithm.

So just to make this more concrete, right, the learning problem in helicopter flight is ten times per second, say. Your sensors on the helicopter gives you a very accurate estimate of the position of orientation of the helicopter and so you know where the helicopter is pretty accurately at all points in time. And your job is to take this input, the position orientation, and to output a set of numbers that correspond to where to move the control sticks to control the helicopter, to make it fly, to right side up, fly upside down, actually whatever maneuver you want.

And this is different from supervised learning because usually we actually don't know what the "right" control action is. And more specifically, if the helicopter is in a certain position orientation, it's actually very hard to say when the helicopter is doing this, you should move the control sticks exactly these positions. So it's very hard to apply supervised learning algorithms to this problem because we can't come up with a training set where the inputs of the position and the output is all the "right" control actions. It's really hard to come up with a training set like that.

Instead of reinforcement learning, we'll give the learning algorithm a different type of feedback, basically called a reward signal, which will tell the helicopter when it's doing well and when it's doing poorly. So what we'll end up doing is we're coming up with something called a reward signal and I'll formalize this later, which will be a measure of how well the helicopter is doing, and then it will be the job of the learning algorithm to take just this reward function as input and try to fly well.

Another good example of reinforcement learning is thinking about getting a program to play a game, to play chess, a game of chess. At any stage in the game, we actually don't know what the "optimal" move is, so it's very hard to pose playing chess as a supervised learning problem because we can't say the x's are the board positions and the y's are the optimum moves because we just don't know how we create any training examples of optimum moves of chess.

But what we do know is if you have a computer playing games of chess, we know when its won a game and when its lost a game, so what we do is we give it a reward signal, so give it a positive reward when it wins a game of chess and give it a negative reward whenever it loses, and hopefully have it learn to win more and more games by itself over time.

So what I'd like you to think about reinforcement learning is think about training a dog. Every time your dog does something good, you sort of tell it, "Good dog," and every time it does something bad, you

tell it, "Bad dog," and over time, your dog learns to do more and more good things over time.

So in the same way, when we try to fly a helicopter, every time the helicopter does something good, you say, "Good helicopter," and every time it crashes, you say, "Bad helicopter," and then over time, it learns to do the right things more and more often.

The reason – one of the reasons that reinforcement learning is much harder than supervised learning is because this is not a one-shot decision making problem. So in supervised learning, if you have a classification, prediction if someone has cancer or not, you make a prediction and then you're done, right? And your patient either has cancer or not, you're either right or wrong, they live or die, whatever. You make a decision and then you're done.

In reinforcement learning, you have to keep taking actions over time, so it's called the sequential decision making. So concretely, suppose a program loses a game of chess on move No. 60. Then it has actually made 60 moves before it got this negative reward of losing a game of chess and the thing that makes it hard for the algorithm to learn from this is called the credit assignment problem. And just to state that informally, what this is if the program loses a game of chess in move 60, you're actually not quite sure of all the moves he made which ones were the right moves and which ones were the bad moves, and so maybe it's because you've blundered on move No. 23 and then everything else you did may have been perfect, but because you made a mistake on move 23 in your game of chess, you eventually end up losing on move 60.

So just to define very loosely for the assignment problem is whether you get a positive or negative reward, so figure out what you actually did right or did wrong to cause the reward so you can do more of the right things and less of the wrong things. And this is sort of one of the things that makes reinforcement learning hard.

And in the same way, if the helicopter crashes, you may not know. And in the same way, if the helicopter crashes, it may be something you did many minutes ago that causes the helicopter to crash. In fact, if you ever crash a car – and hopefully none of you ever get in a car accident – but when someone crashes a car, usually the things they're doing right before they crash is step on the brakes to slow the car down before the impact. And usually stepping on the brakes does not cause a crash. Rather it makes the crash sort of hurt less.

But so, reinforcement algorithm, you see this pattern in that you step on the brakes, you crash, it's not the reason you crash and it's hard to figure out that it's not actually your stepping on the brakes that caused the crash, but something you did long before that.

So let me go ahead and define the – formalize the reinforcement learning problem more, and as a preface, let me say algorithms are applied to a broad range of problems, but because robotics videos are easy to show in the lecture – I have a lot of them – throughout this lecture I use a bunch of robotics for examples, but later, we'll talk about applications of these ideas, so broader ranges of problems as well. But the basic problem I'm facing is sequential decision making. We need to make many decisions and where your decisions perhaps have long term consequences.

So let's formalize the reinforcement learning problem. Reinforcement learning problems model the worlds using something called the MDP or the Markov Decision Process formalism. And let's see, MDP is a five tuple – I don't have enough space – well, comprising five things.

So let me say what these are. Actually, could you please raise the screen? I won't need the laptop anymore today. [Inaudible] more space. Yep, go, great. Thanks.

So an MDP comprises a five tuple. The first of these elements, s is a set of states and so for the helicopter example, the set of states would be the possible positions and orientations of a helicopter. A is a set of actions. So again, for the helicopter example, this would be the set of all possible positions that we could put our control sticks into. P, s, a are state transition distributions. So for each state and each action, this is a high probability distribution, so sum over s prime, Psa, s prime equals 1 and Psa s prime is created over zero.

And state transition distributions are – or state transition probabilities work as follows. P subscript (s, a) gives me the probability distribution over what state I will transition to, what state I wind up in, if I take an action a in a state s. So this is probability distribution over states s prime and then I get to when I take an action a in the state s. Now I'll read this in a second.

Gamma is the number called the discount factor. Don't worry about this yet. I'll say what this is in a second. And there's usually a number strictly rated, strictly less than 1 and rated equal to zero. And R is our reward function, so the reward function maps from the set of states to the real numbers and can be positive or negative. This is the set of real numbers.

So just to make these elements concrete, let me give a specific example of a MDP. Rather than talking about something as complicated as helicopters, I'm going to use a much smaller MDP as the running example for the rest of today's lecture. We'll look at much more complicated MDPs in subsequent lectures.

This is an example that I adapted from a textbook by Stuart Russell and Peter Norvig called Artificial Intelligence: A Modern Approach (Second Edition). And this is a small MDP that models a robot navigation task in which if you imagine you have a robot that lives all over the grid world where the shaded-in cell is an obstacle, so the robot can't go over this cell.

And so, let's see. I would really like the robot to get to this upper right north cell, let's say, so I'm going to associate that cell with a +1 reward, and I'd really like it to avoid that grid cell, so I'm gonna associate that grid cell with -1 reward.

So let's actually iterate through the five elements of the MDP and so see what they are for this problem. So the robot can be in any of these eleven positions and so I have an MDP with 11 states, and it's a set capital S corresponding to the 11 places it could be in. And let's say my robot in this set, highly simplified for a logical example, can try to move in each of the compass directions, so in this MDP, I'll have four actions corresponding to moving in each of the North, South, East, West compass directions.

And let's see. Let's say that my robot's dynamics are noisy. If you've worked in robotics before, you know that if you command a robot to go North, because of wheel slip or a core design in how you act or whatever, there's a small chance that your robot will be off side here. So you command your robot to move forward one meter, usually it will move forward somewhere between like 95 centimeters or to 105 centimeters.

So in this highly simplified grid world, I'm going to model the stochastic dynamics of my robot as follows. I'm going to say that if you command the robot to go north, there's actually a 10 percent chance that it will accidentally veer off to the left and a 10 percent chance it will veer off to the right and only a .8 chance that it will manage to go in the direction you commanded it. This is sort of a crude model, wheels slipping on the model robot. And if the robot bounces off a wall, then it just stays where it is and nothing happens.

So let's see. Concretely, we would write this down using the state transition probability. So for example, let's take the state – let me call it a free comma one state and let's say you command the robot to go north. To specify these noisy dynamics of the robot, you would write down state transition probabilities for the robot as follows. You say that if you're in the state free one and you take the action north, your chance of getting to free two is 0.8. If you're in the state of free one and you take the action north, the chance of getting to four 1 is open 1 and so on. And so on, okay?

This last line is that if you're in the state free one and you take the action north, the chance of you getting to the state free free is zero. And this is your chance of transitioning in one-time sets of the state free free is equal to zero. So these are the state transition probabilities for my MDP.

Let's see. The last two elements of my five tuple are gamma and the reward function. Let's not worry about gamma for now, but my reward function would be as follows, so I really want the robot to get to the fourth – I'm using four comma free. It's indexing to the states by using the numbers I wrote at the sides of the grid.

So my reward for getting to the fourth free state is +1 and my reward for getting to the fourth 2-state is -1, and as is common practice – let's see. As is fairly common practice in navigation tasks, for all other states, the terminal states, I'm going to associate sort of a small negative reward and you can think of this as a small negative reward that charges my robot for his battery consumption or his fuel consumption for one move around. And so a small negative reward like this that charges the robot for running around randomly tends to cause the system to compute solutions that don't waste time and make its way to the goal as quickly as possible because it's charged for fuel consumption.

Okay. So, well, let me just mention, there's actually one other complication that I'm gonna sort of not worry about. In this specific example, unless you're going to assume that when the robot gets to the +1 or the -1 reward, then the world ends and so you get to the +1 and then that's it. The world ends. There

are no more rewards positive or negative after that, right? And so there are various ways to model that. One way to think about that is you may imagine there's actually a 12th state, something that's called the zero cost absorbing state, so that whenever you get to the +1 or the -1, you then transition the probability one to this 12th state and you stay in this 12th state forever with no more rewards. I just mention that, that when you get to the +1 or -1, think of the problems in finishing. The reason I do that is because it makes some of the numbers come up nicer and be easier to understand. It's the sort of state where you go in where sometimes you hear the term zero cost absorbing states. It's another state so that when you enter that state, there are no more rewards; you always stay in that state forever.

All right. So let's just see how an MDP works and it works as follows. At time 0, your robot starts off at some state as 0 and, depending on where you are, you get to choose an action a0 to decide to go North, South, East, or West. Depending on your choice, you get to some state s1, which is going to be randomly drawn from the state transition distribution index by state 0 and the action you just chose. So the next state you get to depends – well, it depends in the probabilistic way on the previous state and the action you just took. After you get to the state s1, you get to choose a new action a1, and then as a result of that, you get to some new state s2 sort of randomly from the state transition distributions and so on. Okay?

So after your robot does this for a while, it will have visited some sequence of states s0, s1, s2, and so on, and to evaluate how well we did, we'll take the reward function and we'll apply it to the sequence of states and add up the sum of rewards that your robot obtained on the sequence of states it visited. State s0 is your action, you get to s1, take an action, you get to s2 and so on. So you keep the reward function in the pile to every state in the sequence and this is the sum of rewards you obtain. Let me show you just one more bit. You can multiply this by gamma, gamma squared, and the next term will be multiplied by gamma cubed and so on. And this is called – I'm going to call this the Total Payoff for the sequence of states s0, s1, s2, and so on that your robot visited. And so let me also say what gamma is. See the quality gamma is a number that's usually less than one. It usually you think of gamma as a number like open 99. So the effect of gamma is that the reward you obtain at time 1 is given a slightly smaller weight than the reward you get at time zero. And then the reward you get at time 2 is even a little bit smaller than the reward you get at a previous time set and so on. Let's see. And so if this is an economic application, if you're in like stock market trading with Gaussian algorithm or whatever, this is an economic application, then your rewards are dollars earned and lost. Then to this kind of factor, gamma has a very natural interpretation as the time value of money because like a dollar today is worth slightly less than – excuse me, the dollar today is worth slightly more than the dollar tomorrow because the dollar in the bank can earn a little bit of interest. And conversely, having to pay out a dollar tomorrow is better than having to pay out a dollar today. So in other words, the effect of this compacted gamma tends to weight wins or losses in the future less than wins or losses in the immediate future – tends to weight wins and losses in the distant future less than wins and losses in the immediate. And so the girth of the reinforcement learning algorithm is to choose actions over time, to choose actions a0, a1 and so on to try to maximize the expected value of this total payoff. And more concretely, what we will try to do is have our reinforcement learning algorithms compute a policy, which I denote by the lower case p, which – and all a policy is, a definition of a policy is a function mapping from the states of the actions and so it goes to kind of a policy that tells us so for every state, what action it recommends we take in that state. So concretely, here is an example of a policy. And this actually turns out to be the optimal policy for the MDP and I'll tell you later how I computed this. And so this is an example of a policy. A policy is just a mapping from the states to the actions, and so our policy tells me when you're in this state, you should take the left action and so on. And this particular policy I drew out happens to be after a policy in the sense that when you execute this policy, this will maximize your expected value of the total payoff. This will maximize your expected total sum of the counter rewards.

Student:[Inaudible]

Instructor (Andrew Ng): Yeah, so can policy be over multiple states, can it be over – so can it be a function of not only current state, but the state I was in previously as well. So the answer is yes. Sometimes people call them strategies instead of policies, but usually you're going to use policies. It actually turns out that for an MDP, you're allowing policies that depend on my previous states, will not allow you to do any better. At least in the limited context we're talking about. So in other words, there exists a policy that only

ever lets the current state ever maximize my expected total payoff. And this statement won't be true for some of the richer models we talk about later, but for now, all we need to do is – this suffices to just look at the current states and actions.

And sometimes they use the term executable policy to mean that I'm going to take actions according to the policies, so I'm going to execute the policy p. That means I'm going to – whenever some state s, I'm going to take the action that the policy p outputs when given the current state.

All right. So it turns out that one of the things MDPs are very good at is – all right, let's look at our states. Say the optimal policy in this state is to go left. There's actually – this probably wasn't very obvious. Why is it that you have actions that go left take a longer path there? The alternative would be to go north and try to find a much shorter path to the +1 state, but when you're in this state over here, this, I guess, free comma 2 state, when in that state over there, when you go north, there's a .1 chance you accidentally veer off to the right to the -1 state. And so there will be subtle tradeoffs. Is it better to take the longer, safer route, but the discount factor tends to discourage that and the .02 charge per step will tend to discourage that. Or is it better to take a shorter, riskier route.

And so it wasn't obvious to me until I computed it, but just to see also action and this is one of those things that MDP machinery is very good at making, to make subtle tradeoffs to make these optimal.

So what I want to do next is make a few more definitions and that will lead us to our first algorithm for computing optimal policies and MDPs, so finding optimal ways to act on MDPs. Before I move on, let's check for any questions about the MDP formalism. Okay, cool.

So let's now talk about how we actually go about computing optimal policy like that and to get there, I need to define a few things. So just as a preview of the next moves I'm gonna take, I'm going to define something called Vp and then I'm going to define V* and then I'm going to define p*. And it will be a consequence of my definitions that p* is the optimal policy.

And so I'm going to say as I define these things, keep in mind what is a definition and what is a consequence of a definition. In particular, I won't be defining p* to be the optimal policy, but I'll define p* by a different equation and it will be a consequence of my definition that p* is the optimal policy.

The first one I want to define is Vp, so for any given policy p, for any policy p, I'm going to define the value function Vp and sometimes I call this the value function for p. So I want to find the value function for Vp, the function mapping from the state's known numbers, such that Vp(s) is the expected payoff – is the expected total payoff if you started in the state s and execute p. So in other words, Vp(s) is equal to the expected value of this here, sum of this counted rewards, the total payoff, given that you execute the policy p and the first state in the sequence is zero, is that state s.

I say this is slightly sloppy probabilistic notation, so p isn't really in around the variable, so maybe I shouldn't actually be conditioning on p. This is sort of moderately standard notation horror, so we use the steady sloppy policy notation.

So as a concrete example, here's a policy and this is not a great policy. This is just some policy p. It's actually a pretty bad policy that for many states, seems to be heading to the -1 rather than the +1. And so the value function is the function mapping from the states of known numbers, so it associates each state with a number and in this case, this is Vp. So that's the value function for this policy.

And so you notice, for instance, that for all the states in the bottom two rows, I guess, this is a really bad policy that has a high chance to take you to the -1 state and so all the values for those states in the bottom two rows are negative. So in this expectation, your total payoff would be negative. And if you execute this rather bad policy, you start in any of the states in the bottom row, and if you start in the top row, the total payoff would be positive. This is not a terribly bad policy if it stays in the topmost row.

And so given any policy, you can write down a value function for that policy. If some of you are still writing, I'll leave that up for a second while I clean another couple of boards.

Okay. So the circumstances of the following, Vp(s) is equal to – well, the expected value of R if s is zero, which is the reward you get right away for just being in the initial state s, plus – let me write this like this – I'm going to write gamma and then R if s1 plus gamma, R of s2 plus dot, dot, dot, what condition of p. Okay? So just de-parenthesize these.

This first term here, this is sometimes called the immediate reward. This is the reward you get right

away just for starting in the state at zero. And then the second term here, these are sometimes called the future reward, which is the rewards you get sort of one time step into the future and what I want you to note is what this term is. That term there is really just the value function for the state s1 because this term here in parentheses, this is really – suppose I was to start in the state s1 or this is the sum of the counter rewards I would get if I were to start in the state s1. So my immediate reward for starting in s1 would be R (s1), then plus gamma times additional future rewards in the future.

And so it turns out you can write VRp recursively in terms of itself. And presume that VRp is equal to the immediate reward plus gamma times – actually, let me write – it would just be mapped as notation – s0 gets mapped to s and s1 gets mapped to s prime and it just makes it all right.

So value function for p to stay zero is the immediate reward plus this current factor gamma times – and now you have Vp of s1. Right here is Vp of s prime. But s prime is a random variable because the next state you get to after one time set is random and so in particular, taking expectations, this is the sum of all states s prime of your probability of getting to that state times that. And just to be clear on this notation, right, this is P subscript (s, a) of s prime is the chance of you getting to the state s prime when you take the action a in state s.

And in this case, we're executing the policy p and so this is P(s) p(s) because the action we're going to take in state s is the action p reverse. So this is – in other words, this Ps p(s), this distribution overstates s prime that you would transitioned to, the one time step, when you take the action p(s) in the state s.

So just to give this a name, this equation is called Bellman's equations and is one of the central equations that we'll use over and over when we solve MDPs. Raise your hand if this equation makes sense. Some of you didn't raise your hands. Do you have questions?

So let's try to say this again. Actually, which of the symbols don't make sense for those of you that didn't raise your hands? You're regretting not raising your hand now, aren't you? Let's try saying this one more time and maybe it will come clear later. So what is it? So this equation is sort of like my value at the current state is equal to R(s) plus gamma times – and depending on what state I get to next, my expected total payoff from the state s prime is Vp of s prime, whereas prime is the state I get to after one time step. So incurring one state as I'm going to take some action and I get to some state that's prime and this equation is sort of my expected total payoff for executing the policy p from the state s.

But s prime is random because the next state I get to is random and well, we'll use the next board. The chance I get to some specific state as prime is given by we have a P subscript (s, a), s prime, where – because these are just [inaudible] and probabilities – where the action a I chose is given by p(s) because I'm executing the action a in the current state s. And so when you plug this back in, you get P subscript s R(s) as prime, just gives me the distribution over the states in making the transition to it in one step, and hence, that just needs Bellman's equations.

So since Bellman's equations gives you a way to solve for the value function for policy in closed form. So again, the problem is suppose I'm given a fixed policy. How do I solve for Vp? How do I solve for the – so given fixed policy, how do I solve for this equation?

It turns out, Bellman's equation gives you a way of doing this, so coming back to this board, it turns out to be – sort of coming back to the previous boards, around – could you move the camera to point to this board? Okay, cool. So going back to this board, we work the problem's equation, this equation, right, let's say I have a fixed policy p and I want to solve for the value function for the policy p. Then what this equation is just imposes a set of linear constraints on the value function. So in particular, this says that the value for a given state is equal to some constant, and then some linear function of other values.

And so you can write down one such equation for every state in your MDP and this imposes a set of linear constraints on what the value function could be. And then it turns out that by solving the resulting linear system equations, you can then solve for the value function Vp(s). There's a high level description. Let me now make this concrete.

So specifically, let me take the free one state, that state we're using as an example. So Bellman's equation tells me that the value for p for the free one state – oh, and let's say I have a specific policy so that p are free one – let's say it takes a North action, which is not the ultimate action. For this policy, Bellman's equation tells me that Vp of free one is equal to R of the state free one, and then plus gamma times our trans 0.8 I

get to the free two state, which translates .1 and gets to the four one state and which times 0.1, I will get to the two one state.

And so what I've done is I've written down Bellman's equations for the free one state. I hope you know what it means. It's in my low MDP; I'm indexing the states 1, 2, 3, 4, so this state over there where I drew the circle is the free one state.

So for every one of my 11 states in the MDP, I can write down an equation like this. This stands just for one state. And you notice that if I'm trying to solve for the values, so these are the unknowns, then I will have 11 variables because I'm trying to solve for the value function for each of my 11 states, and I will have 11 constraints because I can write down 11 equations of this form, one such equation for each one of my states.

So if you do this, if you write down this sort of equation for every one of your states and then do these, you have your set of linear equations with 11 unknowns and 11 variables – excuse me, 11 constraints or 11 equations with 11 unknowns, and so you can solve that linear system of equations to get an explicit solution for Vp. So if you have n states, you end up with n equations and n unknowns and solve that to get the values for all of your states.

Okay, cool. So actually, could you just raise your hand if this made sense? Cool.

All right, so that was the value function for specific policy and how to solve for it. Let me define one more thing. So the optimal value function when defined as V*(s) equals max over all policies p of Vp(s). So in other words, for any given state s, the optimal value function says suppose I take a max over all possible policies p, what is the best possible expected – some of the counter rewards that I can expect to get? Or what is my optimal expected total payoff for starting at state s, so taking a max over all possible control policies p.

So it turns out that there's a version of Bellman's equations for V* as well and so this is also called Bellman's equations for V* rather than for Vp and I'll just write that down. So this says that the optimal payoff you can get from the state s is equal to – so our [inaudible] are multi here, so let's see. Just for starting off in the state s, you're going to get your immediate R(s) and then depending on what action a you take your expected total payoff will be given by this. So if I take an action a in some state s, then with probability given by P subscript (s; a) of s prime, by this probability our transition of state s prime, and when we get to the state s prime, I'll expect my total payoff from there to be given by V*(s) prime because I'm now starting to use the s prime.

So the only thing in this equation I need to fill in is where is the action a, so in order to actually obtain the optimal expected payoff, and to actually obtain the maximum or the optimal expected total payoff, what you should choose here is the max over our actions a, choose your action a that maximizes the expected value of your total payoffs as well.

So it just makes sense. There's a version of Bellman's equations for V* rather than Vp and I'll just say it again. It says that my optimal expected total payoff is my immediate reward plus, and then the best action it can choose, the max over all actions a of my expected future payoff.

And these also lead to my definition of p*, which is let's say I'm in some state s and I want to know what action to choose. Well, if I'm in some state s, I'm gonna get here an immediate R(s) anyway, so what's the best action for me to choose is whatever action will enable me to maximize the second term, as well as if my robot is in some state s and it wants to know what action to choose, I want to choose the action that will maximize my expected total payoff and so p*(s) is going to define as R(max) over actions a of this same thing.

I could also put the gamma there, but gamma is just a positive. Gamma is almost always positive, so I just drop that because it's just a constant scale you go through and doesn't affect the R(max).

And so, the consequence of this definition is that p* is actually the optimal policy because p* will maximize my expected total payoffs.

Cool. Any questions at this point? Cool. So what I'd like to do now is talk about how algorithms actually compute high start, compute the optimal policy. I should write down a little bit more before I do that, but notice that if I can compute V*, if I can compute the optimal value function, then I can plug it into this equation and then I'll be done. So if I can compute V*, then you are using this definition for p*

and can compute the optimal policy.

So my strategy for computing the optimal policy will be to compute V* and then plug it into this equation and that will give me the optimal policy p*. So my goal, my next goal, will really be to compute V*.

But the definition of V* here doesn't lead to a nice algorithm for computing it because let's see − so I know how to compute Vp for any given policy p by solving that linear system equation, but there's an exponentially large number of policies, so you get 11 states and four actions and what the number of policies is froze to the par of 11. This is of a huge space of possible policies and so I can't actually exhaust the union of all policies and then take a max on [inaudible].

So I should write down some other things first, just to ground the notations, but what I'll do is eventually come up with an algorithm for computing V*, the optimal value function and then we'll plug them into this and that will give us the optimal policy p*.

And so I'll write down the algorithm in a second, but just to ground the notation, well − yeah, let's skip that. Let's just talk about the algorithm. So this is an algorithm called value iteration and it makes use of Bellman's equations for the optimal policy to compute V*. So here's the algorithm. Okay, and that's the entirety of the algorithm and oh, you repeat the step, I guess. You repeatedly do this step.

So just to be concrete, let's say in my MDP of 11 states, the first step is initialize V(s) equals zero, so what that means is I create an array in computer implementation, create an array of 11 elements and say set all of them to zero. Says I can initialize into anything. It doesn't really matter.

And now what I'm going to do is I'll take Bellman's equations and we'll keep on taking the right hand side of Bellman's equations and overwriting and start copying down the left hand side. So we'll essentially iteratively try to make Bellman's equations hold true for the numbers V(s) that are stored along the way. So V(s) here is in the array of 11 elements and I'm going to repeatedly compute the right hand side and copy that onto V(s).

And it turns out that when you do this, this will make V(s) converge to V*(s), so it may be of no surprise because we know V* [inaudible] set inside Bellman's equations.

Just to tell you, some of these ideas that they get more than the problem says, so I won't prove the conversions of this algorithm. Some implementation details, it turns out there's two ways you can do this update. One is when I say for every state s that has performed this update, one way you can do this is for every state s, you can compute the right hand side and then you can simultaneously overwrite the left hand side for every state s. And so if you do that, that's called a sequence update. Right and sequence [inaudible], so update all the states s simultaneously.

And if you do that, it's sometimes written as follows. If you do synchronous update, then it's as if you have some value function, you're at the Ith iteration or Tth iteration of the algorithm and then you're going to compute some function of your entire value function, and then you get to set your value function to your new version, so simultaneously update all 11 values in your s space value function.

So it's sometimes written like this. My B here is called the Bellman backup operator, so the synchronized valuation you sort of take the value function, you apply the Bellman backup operator to it and then the Bellman backup operator just means computing the right hand side of this for all the states and you've overwritten your entire value function.

The only way of performing these updates is asynchronous updates, which is where you update the states one at a time. So you go through the states in some fixed order, so would update V(s) for state No. 1 and then I would like to update V(s) for state No. 2, then state No. 3, and so on. And when I'm updating V(s) for state No. 5, if V(s) prime, if I end up using the values for states 1, 2, 3, and 4 on the right hand side, then I'd use my recently updated values on the right hand side. So as you update sequentially, when you're updating in the fifth state, you'd be using values, new values, for states 1, 2, 3, and 4. And that's called an asynchronous update.

Other versions will cause V(s) conversion to be *(s). In synchronized updates, it makes them just a tiny little bit faster [inaudible] and then it turns out the analysis of value iterations synchronous updates are also easier to analyze and that just matters [inaudible]. Asynchronous has been just a little bit faster.

So when you run this algorithm on the MDP − I forgot to say all these values were computed with gamma equals open 99 and actually, Roger Gross, who's a, I guess, master [inaudible] helped me with computing

some of these numbers. So you compute it. That way you run value relation on this MDP. The numbers you get for V* are as follows: .86, .90 − again, the numbers sort of don't matter that much, but just take a look at it and make sure it intuitively makes sense.

And then when you plug those in to the formula for computing, that I wrote down earlier, for computing p* as a function of V*, then − well, I drew this previously, but here's the optimal policy p*.

And so, just to summarize, the process is run value iteration to compute V*, so this would be this table of numbers, and then I use my form of p* to compute the optimal policy, which is this policy in this case.

Now, to be just completely concrete, let's look at that free one state again. Is it better to go left or is it better to go north? So let me just illustrate why I'd rather go left than north. In the form of the p*, if I go west, then sum over s prime, P(s, a) s prime, P*(sp), this would be − well, let me just write this down. Right, if I go north, then it would be because of that. I wrote it down really quickly, so it's messy writing. The way I got these numbers is suppose I'm in this state, in this free one state. If I choose to go west and with chance .8, I get to .75 − to this table − .75. With chance .1, I veer off and get to the .69, then at chance .1, I go south and I bounce off the wall and I stay where I am.

So that's why my expected future payoff for going west is .8 times .75, plus .1 times .69, plus .1 times .71, the last .71 being if I bounce off the wall to the south and then seeing where I am, that gives you .740.

You can then repeat the same process to estimate your expected total payoff if you go north, so if you do that, with a .8 chance, you end up going north, so you get .69. With a .1 chance, you end up here and .1 chance you end up there. This map leads mentally to that expression and compute the expectation, you get .676. And so your total payoff is higher if you go west − your expected total payoff is higher if you go west than if you go north. And that's why the optimal action in this state is to go west.

So that was value iteration. It turns out there are two sort of standard algorithms for computing optimal policies in MDPs. Value iteration is one. As soon as you finish the writing. So value iteration is one and the other sort of standard algorithm for computing optimal policies in MDPs is called policy iteration. And let me − I'm just going to write this down.

In policy iteration, we initialize the policy p randomly, so it doesn't matter. It can be the policy that always goes north or the policy that takes actions random or whatever. And then we'll repeatedly do the following. Okay, so that's the algorithm.

So the algorithm has two steps. In the first step, we solve. We take the current policy p and we solve Bellman's equations to obtain Vp. So remember, earlier I said if you have a fixed policy p, then yeah, Bellman's equation defines this system of linear equations with 11 unknowns and 11 linear constraints. And so you solve that linear system equation so you get the value function for your current policy p, and by this notation, I mean just let V be the value function for policy p.

Then the second step is you update the policy. In other words, you pretend that your current guess V from the value function is indeed the optimal value function and you let p(s) be equal to that out max formula, so as to update your policy p.

And so it turns out that if you do this, then V will converge to V* and p will converge to p*, and so this is another way to find the optimal policy for MDP.

In terms of tradeoffs, it turns out that − let's see − in policy iteration, the computationally expensive step is this one. You need to solve this linear system of equations. You have n equations and n unknowns, if you have n states. And so if you have a problem with a reasonably few number of states, if you have a problem with like 11 states, you can solve the linear system equations fairly efficiently, and so policy iteration tends to work extremely well for problems with smallish numbers of states where you can actually solve those linear systems of equations efficiently.

So if you have a thousand states, anything less than that, you can solve a system of a thousand equations very efficiently, so policy iteration will often work fine. If you have an MDP with an enormous number of states, so we'll actually often see MDPs with tens of thousands or hundreds of thousands or millions or tens of millions of states. If you have a problem with 10 million states and you try to apply policy iteration, then this step requires solving the linear system of 10 million equations and this would be computationally expensive. And so for these really, really large MDPs, I tend to use value iteration.

Let's see. Any questions about this?

Student: So this is a convex function where – that it could be good in local optimization scheme.

Instructor (Andrew Ng): Ah, yes, you're right. That's a good question: Is this a convex function? It actually turns out that there is a way to pose a problem of solving for V* as a convex optimization problem, as a linear program. For instance, I can break down the solution – you write down V* as a solution, so linear would be the only problem you can solve. Policy iteration converges as gamma T conversion. We're not just stuck with local optimal, but the proof of the conversions of policy iteration sort of uses somewhat different principles in convex optimization. At least the versions as far as I can see, yeah. You could probably relate this back to convex optimization, but not understand the principle of why this often converges.

The proof is not that difficult, but it is also sort of longer than I want to go over in this class. Yeah, that was a good point. Cool. Actually, any questions for any of these?

Okay, so we now have two algorithms for solving MDP. There's a given, the five tuple, given the set of states, the set of actions, the state transition properties, the discount factor, and the reward function, you can now apply policy iteration or value iteration to compute the optimal policy for the MDP.

The last thing I want to talk about is what if you don't know the state transition probabilities, and sometimes you won't know the reward function R as well, but let's leave that aside. And so for example, let's say you're trying to fly a helicopter and you don't really know in advance what state your helicopter will transition to and take an action in a certain state, because helicopter dynamics are kind of noisy. You sort of often don't really know what state you end up in.

So the standard thing to do, or one standard thing to do, is then to try to estimate the state transition probabilities from data. Let me just write this out. It turns out that the MDP has its 5 tuple, right? S, A; you have the transition probabilities, gamma, and R. S and A you almost always know. The state space is sort of up to you to define. What's the state space at the very bottom, factor you're trying to control, whatever. Actions is, again, just one of your actions. Usually, we almost always know these. Gamma, the discount factor is something you choose depending on how much you want to trade off current versus future rewards. The reward function you usually know. There are some exceptional cases. Usually, you come up with a reward function and so you usually know what the reward function is. Sometimes you don't, but let's just leave that aside for now and the most common thing for you to have to learn are the state transition probabilities. So we'll just talk about how to learn that. So when you don't know state transition probabilities, the most common thing to do is just estimate it from data. So what I mean is imagine some robot – maybe it's a robot roaming around the hallway, like in that grid example – you would then have the robot just take actions in the MDP and you would then estimate your state transition probabilities P subscript (s, a) s prime to be – pretty much exactly what you'd expect it to be.

This would be the number of times you took action a in state s and you got to s prime, divided by the number of times you took action a in state s. Okay? So the estimate of this is just all the times you took the action a in the state s, what's the fraction of times you actually got to the state s prime. It's pretty much exactly what you expect it to be. Or you can – or in case you've never actually tried action a in state s, so if this turns out to be 0 over 0, you can then have some default estimate for those vector uniform distribution over all states, this reasonable default.

And so, putting it all together – and by the way, it turns out in reinforcement learning, in most of the earlier parts of this class where we did supervised learning, I sort of talked about the logistic regression algorithm, so it does the algorithm and most implementations of logistic regression – like a fairly standard way to do logistic regression were SVMs or faster analysis or whatever. It turns out in reinforcement learning there's more of a mix and match sense, I guess, so there are often different pieces of different algorithms you can choose to use. So in some of the algorithms I write down, there's sort of more than one way to do it and I'm sort of giving specific examples, but if you're faced with an AI problem, some of you in control of robots, you want to plug in value iteration here instead of policy iteration. You want to do something slightly different than one of the specific things I wrote down. That's actually fairly common, so just in reinforcement learning, there's sort of other major ways to apply different algorithms and mix and match different algorithms. And this will come up again in the weekly lectures. So just putting the things I said together, here would be a – now this would be an example of how you might estimate the state transition probabilities in a MDP and find the policy for it. So you might repeatedly do the following. Let's see.

Take actions using some policy p to get experience in the MDP, meaning that just execute the policy p observed state transitions. Based on the data you get, you then update estimates of your state transition probabilities P subscript (s, a) based on the experience of the observations you just got. Then you might solve Bellman's equations using value iterations, which I'm abbreviating to VI, and by Bellman's equations, I mean Bellman's equations for V*, not for Vp. Solve Bellman's equations using value iteration to get an estimate for P* and then you update your policy by events equals [inaudible].

And now you have a new policy so you can then go back and execute this policy for a bit more of the MDPs to get some more observations of state transitions, get the noisy ones in MDP, use that update to estimate your state transition probabilities again; use value iteration or policy iteration to solve for [inaudible] the value function, get a new policy and so on. Okay? And it turns out when you do this, I actually wrote down value iteration for a reason. It turns out in the third step of the algorithm, if you're using value iteration rather than policy iteration, to initialize value iteration, if you use your solution from the previous used algorithm, right, then that's a very good initialization condition and this will tend to converge much more quickly because value iteration tries to solve for V(s) for every state s. It tries to estimate V*(s) and the s from the * in V(s) and so if you're looking through this and you initialize your value iteration algorithm using the values you have from the previous round through this, then that will often make this converge faster.

But again, this is again here, you can also adjust a small part in policy iteration in here as well and whatever, and this is a fairly typical example of how you would solve a policy, correct digits and then key in and try to find a good policy for a problem for which you did not know the state transition probabilities in advance.

Cool. Questions about this? Cool. So that sure was exciting. This is like our first two MDP algorithms in just one lecture. All right, let's close for today. Thanks.

[End of Audio]

Duration: 73 minutes

1.17 Machine Learning Lecture 17[17]

Figure 1.17

MachineLearning-Lecture17

Instructor (Andrew Ng):Okay, good morning. Welcome back. So I hope all of you had a good Thanksgiving break. After the problem sets, I suspect many of us needed one. Just one quick announcement so as I announced by email a few days ago, this afternoon we'll be doing another tape ahead of lecture, so I won't physically be here on Wednesday, and so we'll be taping this Wednesday's lecture ahead of time. If you're free this afternoon, please come to that; it'll be at 3:45 p.m. in the Skilling Auditorium in Skilling 193 at 3:45. But of course, you can also just show up in class as usual at the usual time or just watch it online as usual also.

Okay, welcome back. What I want to do today is continue our discussion on Reinforcement Learning in MDPs. Quite a long topic for me to go over today, so most of today's lecture will be on continuous state MDPs, and in particular, algorithms for solving continuous state MDPs, so I'll talk just very briefly about discretization. I'll spend a lot of time talking about models, assimilators of MDPs, and then talk about one algorithm called fitted value iteration and two functions which builds on that, and then hopefully, I'll have time to get to a second algorithm called, approximate policy iteration

---
[17]This content is available online at <http://cnx.org/content/m45944/1.3/>.

Just to recap, right, in the previous lecture, I defined the Reinforcement Learning problem and I defined MDPs, so let me just recap the notation. I said that an MDP or a Markov Decision Process, was a ? tuple, comprising those things and the running example of those using last time was this one right, adapted from the Russell and Norvig AI textbook. So in this example MDP that I was using, it had 11 states, so that's where S was. The actions were compass directions: north, south, east and west.

The state transition probability is to capture chance of your transitioning to every state when you take any action in any other given state and so in our example that captured the stochastic dynamics of our robot wondering around [inaudible], and we said if you take the action north and the south, you have a .8 chance of actually going north and .1 chance of veering off, so that .1 chance of veering off to the right so said model of the robot's noisy dynamic with a [inaudible] and the reward function was that $+/-1$ at the absorbing states and -0.02 elsewhere. This is an example of an MDP, and that's what these five things were. Oh, and I used a discount factor G of usually a number slightly less than one, so that's the 0.99. And so our goal was to find the policy, the control policy and that's at ?, which is a function mapping from the states of the actions that tells us what action to take in every state, and our goal was to find a policy that maximizes the expected value of our total payoff. So we want to find a policy. Well, let's see. We define value functions Vp (s) to be equal to this. We said that the value of a policy ? from State S was given by the expected value of the sum of discounted rewards, conditioned on your executing the policy ? and you're stating off your [inaudible] to say in the State S, and so our strategy for finding the policy was sort of comprised of two steps. So the goal is to find a good policy that maximizes the suspected value of the sum of discounted rewards, and so I said last time that one strategy for finding the [inaudible] of a policy is to first compute the optimal value function which I denoted V*(s) and is defined like that. It's the maximum value that any policy can obtain, and for example, the optimal value function for that MDP looks like this. So in other words, starting from any of these states, what's the expected value of the sum of discounted rewards you get, so this is V*. We also said that once you've found V*, you can compute the optimal policy using this.

And so once you've found V*, we can use this equation to find the optimal policy ?* and the last piece of this algorithm was Bellman's equations where we know that V*, the optimal sum of discounted rewards you can get for State S, is equal to the immediate reward you get just for starting off in that state +G(for the max over all the actions you could take)(your future sum of discounted rewards)(your future payoff starting from the State S(p) which is where you might transition to after 1(s). And so this gave us a value iteration algorithm, which was essentially V.I.

I'm abbreviating value iteration as V.I., so in the value iteration algorithm, in V.I., you just take Bellman's equations and you repeatedly do this. So initialize some guess of the value functions. Initialize a zero as the sum rounding guess and then repeatedly perform this update for all states, and I said last time that if you do this repeatedly, then V(s) will converge to the optimal value function, V*(s) and then having found V*(s), you can compute the optimal policy ?*.

Just one final thing I want to recap was the policy iteration algorithm in which we repeat the following two steps. So let's see, given a random initial policy, we'll solve for Vp. We'll solve for the value function for that specific policy. So this means for every state, compute the expected sum of discounted rewards for if you execute the policy ? from that state, and then the other step of policy iteration is having found the value function for your policy, you then update the policy pretending that you've already found the optimal value function, V*, and then you repeatedly perform these two steps where you solve for the value function for your current policy and then pretend that that's actually the optimal value function and solve for the policy given the value function, and you repeatedly update the value function or update the policy using that value function. And last time I said that this will also cause the estimated value function V to converge to V* and this will cause p to converge to ?*, the optimal policy.

So those are based on our last lecture [inaudible] MDPs and introduced a lot of new notation symbols and just summarize all that again. What I'm about to do now, what I'm about to do for the rest of today's lecture is actually build on these two algorithms so I guess if you have any questions about this piece, ask now since I've got to go on please. Yeah.

Student:[Inaudible] how those two algorithms are very different?

Instructor (Andrew Ng):I see, right, so yeah, do you see that they're different? Okay, how it's

different. Let's see. So well here's one difference. I didn't say this 'cause no longer use it today. So value iteration and policy iteration are different algorithms. In policy iteration in this step, you're given a fixed policy, and you're going to solve for the value function for that policy and so you're given some fixed policy ?, meaning some function mapping from the state's actions. So give you some policy and whatever. That's just some policy; it's not a great policy. And in that step that I circled, we have to find the ? of S which means that for every state you need to compute your expected sum of discounted rewards or if you execute this specific policy and starting off the MDP in that state S.

So I showed this last time. I won't go into details today, so I said last time that you can actually solve for Vp by solving a linear system of equations. There was a form of Bellman's equations for Vp, and it turned out to be, if you write this out, you end up with a linear system of 11 equations of 11 unknowns and so you can actually solve for the value function for a fixed policy by solving like a system of linear equations with 11 variables and 11 constraints, and so that's policy iteration; whereas, in value iteration, going back on board, in value iteration you sort of repeatedly perform this update where you update the value of a state as the [inaudible]. So I hope that makes sense that the algorithm of these is different.

Student:[Inaudible] on the atomic kits so is the assumption that we can never get out of those states?

Instructor (Andrew Ng): Yes. There's always things that you where you solve for this [inaudible], for example, and make the numbers come up nicely, but I don't wanna spend too much time on them, but yeah, so the assumption is that once you enter the absorbing state, then the world ends or there're no more rewards after that and you can think of another way to think of the absorbing states which is sort of mathematically equivalent. You can think of the absorbing states as transitioning with probability 1 to sum 12 state, and then once you're in that 12th state, you always remain in that 12th state, and there're no further rewards from there. If you want, you can think of this as actually an MDP with 12 states rather than 11 states, and the 12th state is this zero cost absorbing state that you get stuck in forever. Other questions? Yeah, please go.

Student: Where did the Bellman's equations [inaudible] to optimal value [inaudible]?

Instructor (Andrew Ng): Boy, yeah. Okay, this Bellman's equations, this equation that I'm pointing to, I sort of tried to give it justification for this last time. I'll say it in one sentence so that's that the expected total payoff I get, I expect to get something from the state as is equal to my immediate reward which is the reward I get for starting a state. Let's see. If I sum the state, I'm gonna get some first reward and then I can transition to some other state, and then from that other state, I'll get some additional rewards from then. So Bellman's equations breaks that sum into two pieces. It says the value of a state is equal to the reward you get right away is really, well. V*(s) is really equal to +G, so this is V*(s) is, and so Bellman's equations sort of breaks V* into two terms and says that there's this first term which is the immediate reward, that, and then +G(the rewards you get in the future) which it turns out to be equal to that second row.

I spent more time justifying this in the previous lecture, although yeah, hopefully, for the purposes of this lecture, if you're not sure where this is came, if you don't remember the justification of that, why don't you just maybe take my word for that this equation holds true since I use it a little bit later as well, and then the lecture notes sort of explain a little further the justification for why this equation might hold true. But for now, yeah, just for now take my word for it that this holds true 'cause we'll use it a little bit later today as well.

Student:[Inaudible] and would it be in sort of turn back into [inaudible].

Instructor (Andrew Ng): Actually, [inaudible] right question is if in policy iteration if we represent ? implicitly, using V(s), would it become equivalent to valuation, and the answer is sort of no. Let's see. It's true that policy iteration and value iteration are closely related algorithms, and there's actually a continuum between them, but yeah, it actually turns out that, oh, no, the algorithms are not equivalent. It's just in policy iteration, there is a step where you're solving for the value function for the policy vehicle is V, solve for Vp. Usually, you can do this, for instance, by solving a linear system of equations. In value iteration, it is a different algorithm, yes. I hope it makes sense that at least cosmetically it's different.

Student:[Inaudible] you have [inaudible] representing ? implicitly, then you won't have to solve that to [inaudible] equations.

Instructor (Andrew Ng): Yeah, the problem is - let's see. To solve for Vp, this works only if you have

a fixed policy, so once you change a value function, if ? changes as well, then it's sort of hard to solve this. Yeah, so later on we'll actually talk about some examples of when ? is implicitly represented but at least for now it's I think there's − yeah. Maybe there's a way to redefine something, see a mapping onto value iteration but that's not usually done. These are viewed as different algorithms.

Okay, cool, so all good questions. Let me move on and talk about how to generalize these ideas to continuous states. Everything we've done so far has been for discrete states or finite-state MDPs. Where, for example, here we had an MDP with a finite set of 11 states and so the value function or V(s) or our estimate for the value function, V(s), could then be represented using an array of 11 numbers 'cause if you have 11 states, the value function needs to assign a real number to each of the 11 states and so to represent V(s) using an array of 11 numbers. What I want to do for [inaudible] today is talk about continuous states, so for example, if you want to control any of the number of real [inaudible], so for example, if you want to control a car, a car is positioned given by XYT, as position and orientation and if you want to Markov the velocity as well, then Xdot, Ydot, Tdot, so these are so depending on whether you want to model the kinematics and so just position, or whether you want to model the dynamics, meaning the velocity as well.

Earlier I showed you video of a helicopter that was flying, using a rain forest we're learning algorithms, so the helicopter which can fly in three-dimensional space rather than just drive on the 2-D plane, the state will be given by XYZ position, FT?, which is ?[inaudible]. The FT? is sometimes used to note the P[inaudible] of the helicopter, just orientation, and if you want to control a helicopter, you pretty much have to model velocity as well which means both linear velocity as well as angular velocity, and so this would be a 12-dimensional state.

If you want an example that is kind of fun but unusual is, and I'm just gonna use this as an example and actually use this little bit example in today's lecture is the inverted pendulum problem which is sort of a long-running classic in reinforcement learning in which imagine that you have a little cart that's on a rail. The rail ends at some point and if you imagine that you have a pole attached to the cart, and this is a free hinge and so the pole here can rotate freely, and your goal is to control the cart and to move it back and forth on this rail so as to keep the pole balanced. Yeah, there's no long pole in this class but you know what I mean, so you can imagine. Oh, is there a long pole here?

Student: Back in the corner.

Instructor (Andrew Ng): Oh, thanks. Cool. So I did not practice this but you can take a long pole and sort of hold it up, balance, so imagine that you can do it better than I can. Imagine these are [inaudible] just moving back and forth to try to keep the pole balanced, so you can actually us the reinforcement learning algorithm to do that. This is actually one of the longstanding classic problems that people [inaudible] implement and play off using reinforcement learning algorithms, and so for this, the states would be X and T, so X would be the position of the cart, and T would be the orientation of the pole and also the linear velocity and the angular velocity of the pole, so I'll actually use this example a couple times.

So to read continuous state space, how can you apply an algorithm like value iteration and policy iteration to solve the MDP to control like the car or a helicopter or something like the inverted pendulum? So one thing you can do and this is maybe the most straightforward thing is, if you have say a two-dimensional continuous state space, a S-1 and S-2 are my state variables, and in all the examples there are I guess between 4-dimensional to 12-dimensional. I'll just draw 2-D here. The most straightforward thing to do would be to take the continuous state space and discretize it into a number of discrete cells.

And I use S-bar to denote they're discretized or they're discrete states, and so you can [inaudible] with this continuous state problem with a finite or discrete set of states and then you can use policy iteration or value iteration to solve for V*(s)-bar and ?*(s)-bar. And if you're robot is then in some state given by that dot, you would then figure out what discretized state it is in. In this case it's in, this discretized dygrid cell that's called S-bar, and then you execute. You choose the policy. You choose the action given by applied to that discrete state, so discretization is maybe the most straightforward way to turn a continuous state problem into a discrete state problem.

Sometimes you can sorta make this work but a couple of reasons why this does not work very well. One reason is the following, and for this picture, let's even temporarily put aside reinforcement learning. Let's just think about doing regression for now and so suppose you have some invariable X and suppose I have

some data, and I want to fill a function. Y is the function of X, so discretization is saying that I'm going to take my horizontal Xs and chop it up into a number of intervals. Sometimes I call these intervals buckets as well. We chop my horizontal Xs up into a number of buckets and then we're approximate this function using something that's piecewise constant in each of these buckets. And just look at this.

This is clearly not a very good representation, right, and when we talk about regression, you just choose some features of X and run linear regression or something. You get a much better fit to the function. And so the sense that discretization just isn't a very good source of piecewise constant functions. This just isn't a very good function for representing many things, and there's also the sense that there's no smoothing or there's no generalization across the different buckets. And in fact, back in regression, I would never have chosen to do regression using this sort of visualization. It's just really doesn't make sense.

And so in the same way, instead of X, V(s), instead of X and some hypothesis function of X, if you have the state here and you're trying to approximate the value function, then you can get discretization to work for many problems but maybe this isn't the best representation to represent a value function. The other problem with discretization and maybe the more serious problem is what's often somewhat fancifully called the curse of dimensionality. And just the observation that if the state space is in RN, and if you discretize each variable into K buckets, so if discretize each variable into K discrete values, then you get on the order of K to the power of N discrete states. In other words, the number of discrete states you end up with grows exponentially in the dimension of the problem, and so for a helicopter with 12-dimensional state space, this would be maybe like 100 to the power of 12, just huge, and it's not feasible. And so discretization doesn't scale well at all with two problems in high-dimensional state spaces, and this observation actually applies more generally than to just robotics and continuous state problems. For example, another fairly well-known applications of reinforcement learning has been to factory automations. If you imagine that you have 20 machines sitting in the factory and the machines lie in a assembly line and they all do something to a part on the assembly line, then they route the part onto a different machine. You want to use reinforcement learning algorithms, [inaudible] the order in which the different machines operate on your different things that are flowing through your assembly line and maybe different machines can do different things. So if you have N machines and each machine can be in K states, then if you do this sort of discretization, the total number of states would be K to N as well. If you have N machines and if each machine can be in K states, then again, you can get this huge number of states. Other well-known examples would be if you have a board game is another example. You'd want to use reinforcement learning to play chess. Then if you have N pieces on your board game, you have N pieces on the chessboard and if each piece can be in K positions, then this is a game sort of the curse of dimensionality thing where the number of discrete states you end up with goes exponentially with the number of pieces in your board game. So the curse of dimensionality means that discretization scales poorly to high-dimensional state spaces or at least discrete representations scale poorly to high-dimensional state spaces. In practice, discretization will usually, if you have a 2-dimensional problem, discretization will usually work great. If you have a 3-dimensional problem, you can often get discretization to work not too badly without too much trouble. With a 4-dimensional problem, you can still often get to where that they could be challenging and as you go to higher and higher dimensional state spaces, the odds and [inaudible] that you need to figure around to discretization and do things like non-uniform grids, so for example, what I've drawn for you is an example of a non-uniform discretization where I'm discretizing S-2 much more finally than S-1. If I think the value function is much more sensitive to the value of state variable S-2 than to S-1, and so as you get into higher dimensional state spaces, you may need to manually fiddle with choices like these with no uniform discretizations and so on. But the folk wisdom seems to be that if you have 2- or 3-dimensional problems, it work fine. With 4-dimensional problems, you can probably get it to work but it'd be just slightly challenging and you can sometimes by fooling around and being clever, you can often push discretization up to let's say about 6-dimensional problems but with some difficulty and problems higher than 6-dimensional would be extremely difficult to solve with discretization. So that's just rough folk wisdom order of managing problems you think about using for discretization. But what I want to spend most of today talking about is [inaudible] methods that often work much better than discretization and which we will approximate V* directly without resulting to these sort of discretizations. Before I jump to the specific representation let me just spend a few minutes talking about the problem setup then. For

today's lecture, I'm going to focus on the problem of continuous states and just to keep things sort of very simple in this lecture, I want view of continuous actions, so I'm gonna see discrete actions A. So it turns out actually that is a critical fact also for many problems, it turns out that the state space is much larger than the states of actions. That just seems to have worked out that way for many problems, so for example, for driving a car the state space is 6-dimensional, so if XY T, Xdot, Ydot, Tdot. Whereas, your action has, you still have two actions. You have forward backward motion and steering the car, so you have 6-D states and 2-D actions, and so you can discretize the action much more easily than discretize the states. The only examples down for a helicopter you've 12-D states in a 4-dimensional action it turns out, and it's also often much easier to just discretize a continuous actions into a discrete sum of actions. And for the inverted pendulum, you have a 4-D state and a 1-D action. Whether you accelerate your cart to the left or the right is one D action and so for the rest of today, I'm gonna assume a continuous state but I'll assume that maybe you've already discretized your actions, just because in practice it turns out that not for all problems, with many problems large actions is just less of a difficulty than large state spaces. So I'm going to assume that we have a model or simulator of the MDP, and so this is really an assumption on how the state transition probabilities are represented. I'm gonna assume and I'm going to use the terms "model" and "simulator" pretty much synonymously, so specifically, what I'm going to assume is that we have a black box and a piece of code, so that I can input any state, input an action and it will output S prime, sample from the state transition distribution. Says that this is really my assumption on the representation I have for the state transition probabilities, so I'll assume I have a box that read take us in for the stated action and output in mixed state. And so since they're fairly common ways to get models of different MDPs you may work with, one is you might get a model from a physics simulator. So for example, if you're interested in controlling that inverted pendulum, so your action is A which is the magnitude of the force you exert on the cart to left or right, and your state is Xdot, T, Tdot. I'm just gonna write that in that order. And so I'm gonna write down a bunch of equations just for completeness but everything I'm gonna write below here is most of what I wanna write is a bit gratuitous, but so since I'll maybe flip open a textbook on physics, a textbook on mechanics, you can work out the equations of motion of a physical device like this, so you find that Sdot. The dot denotes derivative, so the derivative of the state with respect to time is given by Xdot, ?-L(B) cost B over M Tdot, B. And so on where A is the force is the action that you exert on the cart. L is the length of the pole. M is the total mass of the system and so on. So all these equations are good uses, just writing them down for completeness, but by flipping over, open like a physics textbook, you can work out these equations and notions yourself and this then gives you a model which can say that S-2+1. You're still one time step later will be equal to your previous state plus [inaudible], so in your simulator or in my model what happens to the cart every 10th of a second, so ? T would be within one second and then so plus ? T times that. And so that'd be one way to come up with a model of your MDP. And in this specific example, I've actually written down deterministic model because and by deterministic I mean that given an action in a state, the next state is not random, so would be an example of a deterministic model where I can compute the next state exactly as a function of the previous state and the previous action or it's a deterministic model because all the probability mass is on a single state given the previous stated action. You can also make this a stochastic model. A second way that is often used to attain a model is to learn one. And so again, just concretely what you do is you would imagine that you have a physical inverted pendulum system as you physically own an inverted pendulum robot. What you would do is you would then initialize your inverted pendulum robot to some state and then execute some policy, could be some random policy or some policy that you think is pretty good, or you could even try controlling yourself with a joystick or something. But so you set the system off in some state as zero. Then you take some action. Here's zero and the game could be chosen by some policy or chosen by you using a joystick tryina control your inverted pendulum or whatever. System would transition to some new state, S-1, and then you take some new action, A-1 and so on. Let's say you do this for two time steps and sometimes I call this one trajectory and you repeat this M times, so this is the first trial of the first trajectory, and then you do this again. Initialize it in some and so on. So you do this a bunch of times and then you would run the learning algorithm to estimate ST+1 as a function of ST and AT. And for sake of completeness, you should just think of this as inverted pendulum problem, so ST+1 is a 4-dimensional vector. ST, AT will be a 4-dimensional vector and that'll be a real number, and

so you might run linear regression 4 times to predict each of these state variables as a function of each of these 5 real numbers and so on.

Just for example, if you say that if you want to estimate your next state ST+1 as a linear function of your previous state in your action and so A here will be a 4 by 4 matrix, and B would be a 4-dimensional vector, then you would choose the values of A and B that minimize this. So if you want your model to be that ST+1 is some linear function of the previous stated action, then you might pose this optimization objective and choose A and B to minimize the sum of squares error in your predictive value for ST+1 as the linear function of ST and AT. I should say that this is one specific example where you're using a linear function of the previous stated action to predict the next state.

Of course, you can also use other algorithms like low [inaudible] weight to linear regression or linear regression with nonlinear features or kernel linear regression or whatever to predict the next state as a nonlinear function of the current state as well, so this is just [inaudible] linear problems. And it turns out that low [inaudible] weight to linear regression is for many robots turns out to be an effective method for this learning problem as well.

And so having learned to model, having learned the parameters A and B, you then have a model where you say that ST+1 is AST plus BAT, and so that would be an example of a deterministic model or having learned the parameters A and B, you might say that ST+1 is equal to AST + BAT + ?T. And so these would be very reasonable ways to come up with either a deterministic or a stochastic model for your inverted pendulum MDP. And so just to summarize, what we have now is a model, meaning a piece of code, where you can input a state and an action and get an ST+1. And so if you have a stochastic model, then to influence this model, you would actually sample ?T from this [inaudible] distribution in order to generate ST+1.

So it actually turns out that in a preview, I guess, in the next lecture it actually turns out that in the specific case of linear dynamical systems, in the specific case where the next state is a linear function of the current stated action, it actually turns out that there're very powerful algorithms you can use. So I'm actually not gonna talk about that today. I'll talk about that in the next lecture rather than right now but turns out that for many problems of inverted pendulum go if you use low [inaudible] weights and linear regressionals and long linear algorithm 'cause many systems aren't really linear. You can build a nonlinear model.

So what I wanna do now is talk about given a model, given a simulator for your MDP, how to come up with an algorithm to approximate the alpha value function piece. Before I move on, let me check if there're questions on this. Okay, cool. So here's the idea. Back when we talked about linear regression, we said that given some inputs X in supervised learning, given the input feature is X, we may choose some features of X and then approximate the type of variable as a linear function of various features of X, and so just do exactly the same thing to approximate the optimal value function, and in particular, we'll choose some features 5-S of a state S.

And so you could actually choose 5-S equals S. That would be one reasonable choice, if you want to approximate the value function as a linear function of the states, but you can also choose other things, so for example, for the inverted pendulum example, you may choose 5-S to be equal to a vector of features that may be [inaudible]1 or you may have Xdot2, Xdot maybe some cross terms, maybe times X, maybe dot2 and so on. So you choose some vector or features and then approximate the value function as the value of the state as is equal to data transfers times the features. And I should apologize in advance; I'm overloading notation here. It's unfortunate. I use data both to denote the angle of the cart of the pole inverted pendulum. So this is known as the angle T but also using T to denote the vector of parameters in my [inaudible] algorithm. So sorry about the overloading notation.

Just like we did in linear regression, my goal is to come up with a linear combination of the features that gives me a good approximation to the value function and this is completely analogous to when we said that in linear regression our estimate, my response there but Y as a linear function a feature is at the input. That's what we have in linear regression. Let me just write down value iteration again and then I'll written down an approximation to value iteration, so for discrete states, this is the idea behind value iteration and we said that V(s) will be updated as R(s) + G [inaudible].

That was value iteration and in the case of continuous states, this would be the replaced by an [inaudible], an [inaudible] over states rather via sum over states. Let me just write this as R(s) + G([inaudible]) and then that sum over T's prime. That's really an expectation with respect to random state as prime drawn from the state transition probabilities piece SA of V(s) prime. So this is a sum of all states S prime with the probability of going to S prime (value), so that's really an expectation over the random state S prime flowing from PSA of that. And so what I'll do now is write down an algorithm called fitted value iteration that's in approximation to this but specifically for continuous states. I just wrote down the first two steps, and then I'll continue on the next board, so the first step of the algorithm is we'll sample. Choose some set of states at random. So sample S-1, S-2 through S-M randomly so choose a set of states randomly and initialize my parameter vector to be equal to zero. This is analogous to in value iteration where I might initialize the value function to be the function of all zeros. Then here's the end view for the algorithm. Got quite a lot to write actually. Let's see. And so that's the algorithm. Let me just adjust the writing. Give me a second. Give me a minute to finish and then I'll step through this. Actually, if some of my handwriting is eligible, let me know. So let me step through this and so briefly explain the rationale. So the hear of the algorithm is - let's see. In the original value iteration algorithm, we would take the value for each state, V(s)I, and we will overwrite it with this expression here. In the original, this discrete value iteration algorithm was to V(s)I and we will set V(s)I to be equal to that, I think. Now we have in the continuous state case, we have an infinite continuous set of states and so you can't discretely set the value of each of these to that. So what we'll do instead is choose the parameters T so that V(s)I is as close as possible to this thing on the right hand side instead. And this is what YI turns out to be. So completely, what I'm going to do is I'm going to construct estimates of this term, and then I'm going to choose the parameters of my function approximator. I'm gonna choose my parameter as T, so that V(s)I is as close as possible to these. That's what YI is, and specifically, what I'm going to do is I'll choose parameters data to minimize the sum of square differences between T [inaudible] plus 5SI. This thing here is just V(s)I because I'm approximating V(s)I is a linear function of 5SI and so choose the parameters data to minimize the sum of square differences. So this is last step is basically the approximation version of value iteration. What everything else above was doing was just coming up with an approximation to this term, to this thing here and which I was calling YI. And so confluently, for every state SI we want to estimate what the thing on the right hand side is and but there's an expectation here. There's an expectation over a continuous set of states, may be a very high dimensional state so I can't compute this expectation exactly. What I'll do instead is I'll use my simulator to sample a set of states from this distribution from this P substrip, SIA, from the state transition distribution of where I get to if I take the action A in the state as I, and then I'll average over that sample of states to compute this expectation. And so stepping through the algorithm just says that for each state and for each action, I'm going to sample a set of states. This S prime 1 through S prime K from that state transition distribution, still using the model, and then I'll set Q(a) to be equal to that average and so this is my estimate for R(s)I + G(this expected value for that specific action A). Then I'll take the maximum of actions A and this gives me YI, and so YI is for S for that. And finally, I'll run really run linear regression which is that last of the set [inaudible] to get V(s)I to be close to the YIs. And so this algorithm is called fitted value iteration and it actually often works quite well for continuous, for problems with anywhere from 6- to 10- to 20-dimensional state spaces if you can choose appropriate features. Can you raise a hand please if this algorithm makes sense? Some of you didn't have your hands up. Are there questions for those, yeah?

Student:
Is there a recess [inaudible] function in this setup?

Instructor (Andrew Ng):Oh, yes. An MDP comprises SA, the state transition probabilities G and R and so for continuous state spaces, S would be like R4 for the inverted pendulum or something. Actions with discretized state transitions probabilities with specifying with the model or the simulator, G is just a real number like .99 and the real function is usually a function that's given to you.

And so the reward function is some function of your 4-dimensional state, and for example, you might choose a reward function to be minus − I don't know. Just for an example of simple reward function, if we want a -1 if the pole has fallen and there it depends on you choose your reward function to be -1 if the inverted pendulum falls over and to find that its angle is greater than $30°$ or something and zero otherwise.

So that would be an example of a reward function that you can choose for the inverted pendulum, but yes, assuming a reward function is given to you so that you can compute R(s)I for any state. Are there other questions?

Actually, let me try asking a question, so everything I did here assume that we have a stochastic simulator. So it turns out I can simply this algorithm if I have a deterministic simulator, but deterministic simulator is given a stated action, my next state is always exactly determined. So let me ask you, if I have a deterministic simulator, how would I change this algorithm? How would I simplify this algorithm?

Student: Lower your samples that you're drawing [inaudible].

Instructor (Andrew Ng): Right, so Justin's going right. If I have a deterministic simulator, all my samples from those would be exactly the same, and so if I have a deterministic simulator, I can set K to be equal to 1, so I don't need to draw K different samples. I really only need one sample if I have a deterministic simulator, so you can simplify this by setting K=1 if you have a deterministic simulator. Yeah?

Student:I guess I'm really confused about the, yeah, we sorta turned this [inaudible] into something that looks like linear state regression or some' you know the data transpose times something that we're used to but I guess I'm a little. I don't know really know what question to ask but like when we did this before we had like discrete states and everything. We were determined with finding this optimal policy and I guess it doesn't look like we haven't said the word policy in a while so kinda difficult.

Instructor (Andrew Ng): Okay, yeah, so [inaudible] matters back to policy but maybe I should just say a couple words so let me actually try to get at some of what maybe what you're saying. Our strategy for finding optimal policy has been to find some way to find V*, find some way to find the optimal value function and then use that to compute ?* and some of approximations of ?*. So far everything I've been doing has been focused on how to find V*. I just want to say one more word. It actually turns out that for linear regression it's often very easy. It's often not terribly difficult to choose some resource of the features.

Choosing features for approximating the value function is often somewhat harder, so because the value of a state is how good is starting off in this state. What is my expected sum of discounted rewards? What if I start in a certain state? And so what the feature of the state have to measure is really how good is it to start in a certain state? And so for inverted pendulum you actually have that states where the poles are vertical and when a cart that's centered on your track or something, maybe better and so you can come up with features that measure the orientation of the pole and how close you are to the center of the track and so on and those will be reasonable features to use to approximate V*. Although in general it is true that choosing features, the value function approximation, is often slightly trickier than choosing good features for linear regression.

Okay and then Justin's questions of so given V*, how do you go back to actually find a policy? In the discrete case, so we have that ?*(s) is equal to all [inaudible] over A of that. So that's again, I used to write this as a sum over states [inaudible]. I'll just write this as an expectation and so then once you find the optimal value function V*, you can then find the optimal policy ?* by computing the [inaudible]. So if you're in a continuous state MDP, then you can't actually do this in advance for every single state because there's an infinite number of states and so you can't actually perform this computation in advance to every single state.

What you do instead is whenever your robot is in some specific state S is only when your system is in some specific state S like your car is at some position orientation or your inverted pendulum is in some specific position, posed in some specific angle T. It's only when your system, be it a factor or a board game or a robot, is in some specific state S that you would then go ahead and compute this augmax, so it's only when you're in some state S that you then compute this augmax, and then you execute that action A and then as a result of your action, your robot would transition to some new state and then so it'll be given that specific new state that you compute as augmax using that specific state S that you're in.

There're a few ways to do it. One way to do this is actually the same as in the inner loop of the fitted value iteration algorithm so because of an expectation of a large number of states, you'd need to sample some set of states from the simulator and then approximate this expectation using an average over your samples, so it's actually as inner loop of the value iteration algorithm. So you could do that. That's sometimes done. Sometimes it can also be a pain to have to sample a set of states to approximate those expectations every

time you want to take an action in your MDP.

Couple of special cases where this can be done, one special case is if you have a deterministic simulator. If it's a deterministic simulator, so in other words, if your similar is just some function, could be a linear or a nonlinear function. If it's a deterministic simulator then the next state, ST+1, is just some function of your previous stated action. If that's the case then this expectation, well, then this simplifies to augmax of A of V* of F of I guess S,A because this is really saying S prime=F(s),A. I switched back and forth between notation; I hope that's okay. S to denote the current state, and S prime to deterministic state versus ST and ST+1 through the current state. Both of these are sorta standing notation and don't mind my switching back and forth between them. But if it's a deterministic simulator you can then just compute what the next state S prime would be for each action you might take from the current state, and then take the augmax of actions A, basically choose the action that gets you to the highest value state.

And so that's one case where you can compute the augmax and we can compute that expectation without needing to sample an average over some sample. Another very common case actually it turns out is if you have a stochastic simulator, but if your similar happens to take on a very specific form of ST+1=F(s)T,AT+?T where this is galsie noise. The [inaudible] is a very common way to build simulators where you model the next state as a function of the current state and action plus some noise and so once specific example would be that sort of mini dynamical system that we talked about with linear function of the current state and action plus galsie noise. In this case, you can approximate augment over A, well.

In that case you take that expectation that you're trying to approximate. The expected value of V* of S prime, we can approximate that with V* of the expected value of S prime, and this is approximation. Expected value of a function is usually not equal to the value of an expectation but it is often a reasonable approximation and so that would be another way to approximate that expectation and so you choose the actions according to watch we do the same formula as I wrote just now. And so this would be a way of approximating this augmax, ignoring the noise in the simulator essentially. And this often works pretty well as well just because many simulators turn out to be the form of some linear or some nonlinear function plus zero mean galsie noise, so and just that ignore the zero mean galsie noise, so that you can compute this quickly.

And just to complete about this, what that is, right, that V* F of SA, this you down rate as data transfers Fi of S prime where S prime=F of SA. Great, so this V* you would compute using the parameters data that you just learned using the fitted value iteration algorithm. Questions about this?

Student:[Inaudible] case, for real-time application is it possible to use that [inaudible], for example for [inaudible].

Instructor (Andrew Ng):Yes, in real-time applications is it possible to sample case phases use [inaudible] expectation. Computers today actually amazingly fast. I'm actually often surprised by how much you can do in real time so the helicopter we actually flying a helicopter using an algorithm different than this? I can't say. But my intuition is that you could actually do this with a helicopter. A helicopter would control at somewhere between 10hz and 50hz. You need to do this 10 times a second to 50 times a second, and that's actually plenty of time to sample 1,000 states and compute this expectation.

They're real difficult, helicopters because helicopters are mission critical, and you do something it's like fast. You can do serious damage and so maybe not for good reasons. We've actually tended to avoid tossing coins when we're in the air, so the ideal of letting our actions be some up with some random process is slightly scary and just tend not to do that. I should say that's prob'ly not a great reason because you average a large number of things here very well fine but just as a maybe overly conservative design choice, we actually don't, tend not to find anything randomized on which is prob'ly being over conservative. It's the choice we made 'cause other things are slightly safer. I think you can actually often do this.

So long as I see a model can be evaluated fast enough where you can sample 100 state transitions or 1,000 state transitions, and then do that at 10hz. They haven't said that. This is often attained which is why we often use the other approximations that don't require your drawing a large sample. Anything else? No, okay, cool. So now you know one algorithm [inaudible] reinforcement learning on continuous state spaces. Then we'll pick up with some more ideas on some even more powerful algorithms, the solving MDPs of continuous state spaces. Thanks. Let's close for today.

[End of Audio]
Duration: 77 minutes

1.18 Machine Learning Lecture 18[18]

Figure 1.18

MachineLearning-Lecture17

Instructor (Andrew Ng): Okay, good morning. Welcome back. So I hope all of you had a good Thanksgiving break. After the problem sets, I suspect many of us needed one. Just one quick announcement so as I announced by email a few days ago, this afternoon we'll be doing another tape ahead of lecture, so I won't physically be here on Wednesday, and so we'll be taping this Wednesday's lecture ahead of time. If you're free this afternoon, please come to that; it'll be at 3:45 p.m. in the Skilling Auditorium in Skilling 193 at 3:45. But of course, you can also just show up in class as usual at the usual time or just watch it online as usual also.

Okay, welcome back. What I want to do today is continue our discussion on Reinforcement Learning in MDPs. Quite a long topic for me to go over today, so most of today's lecture will be on continuous state MDPs, and in particular, algorithms for solving continuous state MDPs, so I'll talk just very briefly about discretization. I'll spend a lot of time talking about models, assimilators of MDPs, and then talk about one algorithm called fitted value iteration and two functions which builds on that, and then hopefully, I'll have time to get to a second algorithm called, approximate policy iteration

Just to recap, right, in the previous lecture, I defined the Reinforcement Learning problem and I defined MDPs, so let me just recap the notation. I said that an MDP or a Markov Decision Process, was a ? tuple, comprising those things and the running example of those using last time was this one right, adapted from the Russell and Norvig AI textbook. So in this example MDP that I was using, it had 11 states, so that's where S was. The actions were compass directions: north, south, east and west.

The state transition probability is to capture chance of your transitioning to every state when you take any action in any other given state and so in our example that captured the stochastic dynamics of our robot wondering around [inaudible], and we said if you take the action north and the south, you have a .8 chance of actually going north and .1 chance of veering off, so that .1 chance of veering off to the right so said model of the robot's noisy dynamic with a [inaudible] and the reward function was that $+/-1$ at the absorbing states and -0.02 elsewhere. This is an example of an MDP, and that's what these five things were. Oh, and I used a discount factor G of usually a number slightly less than one, so that's the 0.99. And so our goal was to find the policy, the control policy and that's at ?, which is a function mapping from the states of the actions that tells us what action to take in every state, and our goal was to find a policy that maximizes the expected value of our total payoff. So we want to find a policy. Well, let's see. We define value functions Vp (s) to be equal to this. We said that the value of a policy ? from State S was given by the expected value of the sum of discounted rewards, conditioned on your executing the policy ? and you're stating off your [inaudible] to say in the State S, and so our strategy for finding the policy was sort of comprised of two steps. So the goal is to find a good policy that maximizes the suspected value of the sum of discounted rewards, and so I said last time that one strategy for finding the [inaudible] of a policy is to first compute the optimal value function which I denoted $V^*(s)$ and is defined like that. It's the maximum value that any policy can obtain, and for example, the optimal value function for that MDP looks like this. So in other

---

[18]This content is available online at <http://cnx.org/content/m45950/1.3/>.

words, starting from any of these states, what's the expected value of the sum of discounted rewards you get, so this is V*. We also said that once you've found V*, you can compute the optimal policy using this.

And so once you've found V*, we can use this equation to find the optimal policy ?* and the last piece of this algorithm was Bellman's equations where we know that V*, the optimal sum of discounted rewards you can get for State S, is equal to the immediate reward you get just for starting off in that state +G(for the max over all the actions you could take)(your future sum of discounted rewards)(your future payoff starting from the State S(p) which is where you might transition to after 1(s). And so this gave us a value iteration algorithm, which was essentially V.I.

I'm abbreviating value iteration as V.I., so in the value iteration algorithm, in V.I., you just take Bellman's equations and you repeatedly do this. So initialize some guess of the value functions. Initialize a zero as the sum rounding guess and then repeatedly perform this update for all states, and I said last time that if you do this repeatedly, then V(s) will converge to the optimal value function, V*(s) and then having found V*(s), you can compute the optimal policy ?*.

Just one final thing I want to recap was the policy iteration algorithm in which we repeat the following two steps. So let's see, given a random initial policy, we'll solve for Vp. We'll solve for the value function for that specific policy. So this means for every state, compute the expected sum of discounted rewards for if you execute the policy ? from that state, and then the other step of policy iteration is having found the value function for your policy, you then update the policy pretending that you've already found the optimal value function, V*, and then you repeatedly perform these two steps where you solve for the value function for your current policy and then pretend that that's actually the optimal value function and solve for the policy given the value function, and you repeatedly update the value function or update the policy using that value function. And last time I said that this will also cause the estimated value function V to converge to V* and this will cause p to converge to ?*, the optimal policy.

So those are based on our last lecture [inaudible] MDPs and introduced a lot of new notation symbols and just summarize all that again. What I'm about to do now, what I'm about to do for the rest of today's lecture is actually build on these two algorithms so I guess if you have any questions about this piece, ask now since I've got to go on please. Yeah.

Student:[Inaudible] how those two algorithms are very different?

Instructor (Andrew Ng):I see, right, so yeah, do you see that they're different? Okay, how it's different. Let's see. So well here's one difference. I didn't say this 'cause no longer use it today. So value iteration and policy iteration are different algorithms. In policy iteration in this step, you're given a fixed policy, and you're going to solve for the value function for that policy and so you're given some fixed policy ?, meaning some function mapping from the state's actions. So give you some policy and whatever. That's just some policy; it's not a great policy. And in that step that I circled, we have to find the ? of S which means that for every state you need to compute your expected sum of discounted rewards or if you execute this specific policy and starting off the MDP in that state S.

So I showed this last time. I won't go into details today, so I said last time that you can actually solve for Vp by solving a linear system of equations. There was a form of Bellman's equations for Vp, and it turned out to be, if you write this out, you end up with a linear system of 11 equations of 11 unknowns and so you can actually solve for the value function for a fixed policy by solving like a system of linear equations with 11 variables and 11 constraints, and so that's policy iteration; whereas, in value iteration, going back on board, in value iteration you sort of repeatedly perform this update where you update the value of a state as the [inaudible]. So I hope that makes sense that the algorithm of these is different.

Student:[Inaudible] on the atomic kits so is the assumption that we can never get out of those states?

Instructor (Andrew Ng):Yes. There's always things that you where you solve for this [inaudible], for example, and make the numbers come up nicely, but I don't wanna spend too much time on them, but yeah, so the assumption is that once you enter the absorbing state, then the world ends or there're no more rewards after that and you can think of another way to think of the absorbing states which is sort of mathematically equivalent. You can think of the absorbing states as transitioning with probability 1 to sum 12 state, and then once you're in that 12th state, you always remain in that 12th state, and there're no further rewards from there. If you want, you can think of this as actually an MDP with 12 states rather than 11 states, and

the 12th state is this zero cost absorbing state that you get stuck in forever. Other questions? Yeah, please go.

Student: Where did the Bellman's equations [inaudible] to optimal value [inaudible]?

Instructor (Andrew Ng): Boy, yeah. Okay, this Bellman's equations, this equation that I'm pointing to, I sort of tried to give it justification for this last time. I'll say it in one sentence so that's that the expected total payoff I get, I expect to get something from the state as is equal to my immediate reward which is the reward I get for starting a state. Let's see. If I sum the state, I'm gonna get some first reward and then I can transition to some other state, and then from that other state, I'll get some additional rewards from then. So Bellman's equations breaks that sum into two pieces. It says the value of a state is equal to the reward you get right away is really, well. V*(s) is really equal to +G, so this is V*(s) is, and so Bellman's equations sort of breaks V* into two terms and says that there's this first term which is the immediate reward, that, and then +G(the rewards you get in the future) which it turns out to be equal to that second row.

I spent more time justifying this in the previous lecture, although yeah, hopefully, for the purposes of this lecture, if you're not sure where this is came, if you don't remember the justification of that, why don't you just maybe take my word for that this equation holds true since I use it a little bit later as well, and then the lecture notes sort of explain a little further the justification for why this equation might hold true. But for now, yeah, just for now take my word for it that this holds true 'cause we'll use it a little bit later today as well.

Student:[Inaudible] and would it be in sort of turn back into [inaudible].

Instructor (Andrew Ng): Actually, [inaudible] right question is if in policy iteration if we represent ? implicitly, using V(s), would it become equivalent to valuation, and the answer is sort of no. Let's see. It's true that policy iteration and value iteration are closely related algorithms, and there's actually a continuum between them, but yeah, it actually turns out that, oh, no, the algorithms are not equivalent. It's just in policy iteration, there is a step where you're solving for the value function for the policy vehicle is V, solve for Vp. Usually, you can do this, for instance, by solving a linear system of equations. In value iteration, it is a different algorithm, yes. I hope it makes sense that at least cosmetically it's different.

Student:[Inaudible] you have [inaudible] representing ? implicitly, then you won't have to solve that to [inaudible] equations.

Instructor (Andrew Ng): Yeah, the problem is - let's see. To solve for Vp, this works only if you have a fixed policy, so once you change a value function, if ? changes as well, then it's sort of hard to solve this. Yeah, so later on we'll actually talk about some examples of when ? is implicitly represented but at least for now it's I think there's − yeah. Maybe there's a way to redefine something, see a mapping onto value iteration but that's not usually done. These are viewed as different algorithms.

Okay, cool, so all good questions. Let me move on and talk about how to generalize these ideas to continuous states. Everything we've done so far has been for discrete states or finite-state MDPs. Where, for example, here we had an MDP with a finite set of 11 states and so the value function or V(s) or our estimate for the value function, V(s), could then be represented using an array of 11 numbers 'cause if you have 11 states, the value function needs to assign a real number to each of the 11 states and so to represent V(s) using an array of 11 numbers. What I want to do for [inaudible] today is talk about continuous states, so for example, if you want to control any of the number of real [inaudible], so for example, if you want to control a car, a car is positioned given by XYT, as position and orientation and if you want to Markov the velocity as well, then Xdot, Ydot, Tdot, so these are so depending on whether you want to model the kinematics and so just position, or whether you want to model the dynamics, meaning the velocity as well.

Earlier I showed you video of a helicopter that was flying, using a rain forest we're learning algorithms, so the helicopter which can fly in three-dimensional space rather than just drive on the 2-D plane, the state will be given by XYZ position, FT?, which is ?[inaudible]. The FT? is sometimes used to note the P[inaudible] of the helicopter, just orientation, and if you want to control a helicopter, you pretty much have to model velocity as well which means both linear velocity as well as angular velocity, and so this would be a 12-dimensional state.

If you want an example that is kind of fun but unusual is, and I'm just gonna use this as an example and actually use this little bit example in today's lecture is the inverted pendulum problem which is sort of

a long-running classic in reinforcement learning in which imagine that you have a little cart that's on a rail. The rail ends at some point and if you imagine that you have a pole attached to the cart, and this is a free hinge and so the pole here can rotate freely, and your goal is to control the cart and to move it back and forth on this rail so as to keep the pole balanced. Yeah, there's no long pole in this class but you know what I mean, so you can imagine. Oh, is there a long pole here?

Student: Back in the corner.

Instructor (Andrew Ng): Oh, thanks. Cool. So I did not practice this but you can take a long pole and sort of hold it up, balance, so imagine that you can do it better than I can. Imagine these are [inaudible] just moving back and forth to try to keep the pole balanced, so you can actually us the reinforcement learning algorithm to do that. This is actually one of the longstanding classic problems that people [inaudible] implement and play off using reinforcement learning algorithms, and so for this, the states would be X and T, so X would be the position of the cart, and T would be the orientation of the pole and also the linear velocity and the angular velocity of the pole, so I'll actually use this example a couple times.

So to read continuous state space, how can you apply an algorithm like value iteration and policy iteration to solve the MDP to control like the car or a helicopter or something like the inverted pendulum? So one thing you can do and this is maybe the most straightforward thing is, if you have say a two-dimensional continuous state space, a S-1 and S-2 are my state variables, and in all the examples there are I guess between 4-dimensional to 12-dimensional. I'll just draw 2-D here. The most straightforward thing to do would be to take the continuous state space and discretize it into a number of discrete cells.

And I use S-bar to denote they're discretized or they're discrete states, and so you can [inaudible] with this continuous state problem with a finite or discrete set of states and then you can use policy iteration or value iteration to solve for V*(s)-bar and ?*(s)-bar. And if you're robot is then in some state given by that dot, you would then figure out what discretized state it is in. In this case it's in, this discretized dygrid cell that's called S-bar, and then you execute. You choose the policy. You choose the action given by applied to that discrete state, so discretization is maybe the most straightforward way to turn a continuous state problem into a discrete state problem.

Sometimes you can sorta make this work but a couple of reasons why this does not work very well. One reason is the following, and for this picture, let's even temporarily put aside reinforcement learning. Let's just think about doing regression for now and so suppose you have some invariable X and suppose I have some data, and I want to fill a function. Y is the function of X, so discretization is saying that I'm going to take my horizontal Xs and chop it up into a number of intervals. Sometimes I call these intervals buckets as well. We chop my horizontal Xs up into a number of buckets and then we're approximate this function using something that's piecewise constant in each of these buckets. And just look at this.

This is clearly not a very good representation, right, and when we talk about regression, you just choose some features of X and run linear regression or something. You get a much better fit to the function. And so the sense that discretization just isn't a very good source of piecewise constant functions. This just isn't a very good function for representing many things, and there's also the sense that there's no smoothing or there's no generalization across the different buckets. And in fact, back in regression, I would never have chosen to do regression using this sort of visualization. It's just really doesn't make sense.

And so in the same way, instead of X, V(s), instead of X and some hypothesis function of X, if you have the state here and you're trying to approximate the value function, then you can get discretization to work for many problems but maybe this isn't the best representation to represent a value function. The other problem with discretization and maybe the more serious problem is what's often somewhat fancifully called the curse of dimensionality. And just the observation that if the state space is in RN, and if you discretize each variable into K buckets, so if discretize each variable into K discrete values, then you get on the order of K to the power of N discrete states. In other words, the number of discrete states you end up with grows exponentially in the dimension of the problem, and so for a helicopter with 12-dimensional state space, this would be maybe like 100 to the power of 12, just huge, and it's not feasible. And so discretization doesn't scale well at all with two problems in high-dimensional state spaces, and this observation actually applies more generally than to just robotics and continuous state problems. For example, another fairly well-known applications of reinforcement learning has been to factory automations. If you imagine that you have 20

machines sitting in the factory and the machines lie in a assembly line and they all do something to a part on the assembly line, then they route the part onto a different machine. You want to use reinforcement learning algorithms, [inaudible] the order in which the different machines operate on your different things that are flowing through your assembly line and maybe different machines can do different things. So if you have N machines and each machine can be in K states, then if you do this sort of discretization, the total number of states would be K to N as well. If you have N machines and if each machine can be in K states, then again, you can get this huge number of states. Other well-known examples would be if you have a board game is another example. You'd want to use reinforcement learning to play chess. Then if you have N pieces on your board game, you have N pieces on the chessboard and if each piece can be in K positions, then this is a game sort of the curse of dimensionality thing where the number of discrete states you end up with goes exponentially with the number of pieces in your board game. So the curse of dimensionality means that discretization scales poorly to high-dimensional state spaces or at least discrete representations scale poorly to high-dimensional state spaces. In practice, discretization will usually, if you have a 2-dimensional problem, discretization will usually work great. If you have a 3-dimensional problem, you can often get discretization to work not too badly without too much trouble. With a 4-dimensional problem, you can still often get to where that they could be challenging and as you go to higher and higher dimensional state spaces, the odds and [inaudible] that you need to figure around to discretization and do things like non-uniform grids, so for example, what I've drawn for you is an example of a non-uniform discretization where I'm discretizing S-2 much more finally than S-1. If I think the value function is much more sensitive to the value of state variable S-2 than to S-1, and so as you get into higher dimensional state spaces, you may need to manually fiddle with choices like these with no uniform discretizations and so on. But the folk wisdom seems to be that if you have 2- or 3-dimensional problems, it work fine. With 4-dimensional problems, you can probably get it to work but it'd be just slightly challenging and you can sometimes by fooling around and being clever, you can often push discretization up to let's say about 6-dimensional problems but with some difficulty and problems higher than 6-dimensional would be extremely difficult to solve with discretization. So that's just rough folk wisdom order of managing problems you think about using for discretization. But what I want to spend most of today talking about is [inaudible] methods that often work much better than discretization and which we will approximate V* directly without resulting to these sort of discretizations. Before I jump to the specific representation let me just spend a few minutes talking about the problem setup then. For today's lecture, I'm going to focus on the problem of continuous states and just to keep things sort of very simple in this lecture, I want view of continuous actions, so I'm gonna see discrete actions A. So it turns out actually that is a critical fact also for many problems, it turns out that the state space is much larger than the states of actions. That just seems to have worked out that way for many problems, so for example, for driving a car the state space is 6-dimensional, so if XY T, Xdot, Ydot, Tdot. Whereas, your action has, you still have two actions. You have forward backward motion and steering the car, so you have 6-D states and 2-D actions, and so you can discretize the action much more easily than discretize the states. The only examples down for a helicopter you've 12-D states in a 4-dimensional action it turns out, and it's also often much easier to just discretize a continuous actions into a discrete sum of actions. And for the inverted pendulum, you have a 4-D state and a 1-D action. Whether you accelerate your cart to the left or the right is one D action and so for the rest of today, I'm gonna assume a continuous state but I'll assume that maybe you've already discretized your actions, just because in practice it turns out that not for all problems, with many problems large actions is just less of a difficulty than large state spaces. So I'm going to assume that we have a model or simulator of the MDP, and so this is really an assumption on how the state transition probabilities are represented. I'm gonna assume and I'm going to use the terms "model" and "simulator" pretty much synonymously, so specifically, what I'm going to assume is that we have a black box and a piece of code, so that I can input any state, input an action and it will output S prime, sample from the state transition distribution. Says that this is really my assumption on the representation I have for the state transition probabilities, so I'll assume I have a box that read take us in for the stated action and output in mixed state. And so since they're fairly common ways to get models of different MDPs you may work with, one is you might get a model from a physics simulator. So for example, if you're interested in controlling that inverted pendulum, so your action is A which is the magnitude of the force you exert on the cart to

left or right, and your state is Xdot, T, Tdot. I'm just gonna write that in that order. And so I'm gonna write down a bunch of equations just for completeness but everything I'm gonna write below here is most of what I wanna write is a bit gratuitous, but so since I'll maybe flip open a textbook on physics, a textbook on mechanics, you can work out the equations of motion of a physical device like this, so you find that Sdot. The dot denotes derivative, so the derivative of the state with respect to time is given by Xdot, ?-L(B) cost B over M Tdot, B. And so on where A is the force is the action that you exert on the cart. L is the length of the pole. M is the total mass of the system and so on. So all these equations are good uses, just writing them down for completeness, but by flipping over, open like a physics textbook, you can work out these equations and notions yourself and this then gives you a model which can say that S-2+1. You're still one time step later will be equal to your previous state plus [inaudible], so in your simulator or in my model what happens to the cart every 10th of a second, so ? T would be within one second and then so plus ? T times that. And so that'd be one way to come up with a model of your MDP. And in this specific example, I've actually written down deterministic model because and by deterministic I mean that given an action in a state, the next state is not random, so would be an example of a deterministic model where I can compute the next state exactly as a function of the previous state and the previous action or it's a deterministic model because all the probability mass is on a single state given the previous stated action. You can also make this a stochastic model. A second way that is often used to attain a model is to learn one. And so again, just concretely what you do is you would imagine that you have a physical inverted pendulum system as you physically own an inverted pendulum robot. What you would do is you would then initialize your inverted pendulum robot to some state and then execute some policy, could be some random policy or some policy that you think is pretty good, or you could even try controlling yourself with a joystick or something. But so you set the system off in some state as zero. Then you take some action. Here's zero and the game could be chosen by some policy or chosen by you using a joystick tryina control your inverted pendulum or whatever. System would transition to some new state, S-1, and then you take some new action, A-1 and so on. Let's say you do this for two time steps and sometimes I call this one trajectory and you repeat this M times, so this is the first trial of the first trajectory, and then you do this again. Initialize it in some and so on. So you do this a bunch of times and then you would run the learning algorithm to estimate ST+1 as a function of ST and AT. And for sake of completeness, you should just think of this as inverted pendulum problem, so ST+1 is a 4-dimensional vector. ST, AT will be a 4-dimensional vector and that'll be a real number, and so you might run linear regression 4 times to predict each of these state variables as a function of each of these 5 real numbers and so on.

Just for example, if you say that if you want to estimate your next state ST+1 as a linear function of your previous state in your action and so A here will be a 4 by 4 matrix, and B would be a 4-dimensional vector, then you would choose the values of A and B that minimize this. So if you want your model to be that ST+1 is some linear function of the previous stated action, then you might pose this optimization objective and choose A and B to minimize the sum of squares error in your predictive value for ST+1 as the linear function of ST and AT. I should say that this is one specific example where you're using a linear function of the previous stated action to predict the next state.

Of course, you can also use other algorithms like low [inaudible] weight to linear regression or linear regression with nonlinear features or kernel linear regression or whatever to predict the next state as a nonlinear function of the current state as well, so this is just [inaudible] linear problems. And it turns out that low [inaudible] weight to linear regression is for many robots turns out to be an effective method for this learning problem as well.

And so having learned to model, having learned the parameters A and B, you then have a model where you say that ST+1 is AST plus BAT, and so that would be an example of a deterministic model or having learned the parameters A and B, you might say that ST+1 is equal to AST + BAT + ?T. And so these would be very reasonable ways to come up with either a deterministic or a stochastic model for your inverted pendulum MDP. And so just to summarize, what we have now is a model, meaning a piece of code, where you can input a state and an action and get an ST+1. And so if you have a stochastic model, then to influence this model, you would actually sample ?T from this [inaudible] distribution in order to generate ST+1.

So it actually turns out that in a preview, I guess, in the next lecture it actually turns out that in the specific case of linear dynamical systems, in the specific case where the next state is a linear function of the current stated action, it actually turns out that there're very powerful algorithms you can use. So I'm actually not gonna talk about that today. I'll talk about that in the next lecture rather than right now but turns out that for many problems of inverted pendulum go if you use low [inaudible] weights and linear regressionals and long linear algorithm 'cause many systems aren't really linear. You can build a nonlinear model.

So what I wanna do now is talk about given a model, given a simulator for your MDP, how to come up with an algorithm to approximate the alpha value function piece. Before I move on, let me check if there're questions on this. Okay, cool. So here's the idea. Back when we talked about linear regression, we said that given some inputs X in supervised learning, given the input feature is X, we may choose some features of X and then approximate the type of variable as a linear function of various features of X, and so just do exactly the same thing to approximate the optimal value function, and in particular, we'll choose some features 5-S of a state S.

And so you could actually choose 5-S equals S. That would be one reasonable choice, if you want to approximate the value function as a linear function of the states, but you can also choose other things, so for example, for the inverted pendulum example, you may choose 5-S to be equal to a vector of features that may be [inaudible]1 or you may have Xdot2, Xdot maybe some cross terms, maybe times X, maybe dot2 and so on. So you choose some vector or features and then approximate the value function as the value of the state as is equal to data transfers times the features. And I should apologize in advance; I'm overloading notation here. It's unfortunate. I use data both to denote the angle of the cart of the pole inverted pendulum. So this is known as the angle T but also using T to denote the vector of parameters in my [inaudible] algorithm. So sorry about the overloading notation.

Just like we did in linear regression, my goal is to come up with a linear combination of the features that gives me a good approximation to the value function and this is completely analogous to when we said that in linear regression our estimate, my response there but Y as a linear function a feature is at the input. That's what we have in linear regression. Let me just write down value iteration again and then I'll written down an approximation to value iteration, so for discrete states, this is the idea behind value iteration and we said that V(s) will be updated as R(s) + G [inaudible].

That was value iteration and in the case of continuous states, this would be the replaced by an [inaudible], an [inaudible] over states rather via sum over states. Let me just write this as R(s) + G([inaudible]) and then that sum over T's prime. That's really an expectation with respect to random state as prime drawn from the state transition probabilities piece SA of V(s) prime. So this is a sum of all states S prime with the probability of going to S prime (value), so that's really an expectation over the random state S prime flowing from PSA of that. And so what I'll do now is write down an algorithm called fitted value iteration that's in approximation to this but specifically for continuous states. I just wrote down the first two steps, and then I'll continue on the next board, so the first step of the algorithm is we'll sample. Choose some set of states at random. So sample S-1, S-2 through S-M randomly so choose a set of states randomly and initialize my parameter vector to be equal to zero. This is analogous to in value iteration where I might initialize the value function to be the function of all zeros. Then here's the end view for the algorithm. Got quite a lot to write actually. Let's see. And so that's the algorithm. Let me just adjust the writing. Give me a second. Give me a minute to finish and then I'll step through this. Actually, if some of my handwriting is eligible, let me know. So let me step through this and so briefly explain the rationale. So the hear of the algorithm is - let's see. In the original value iteration algorithm, we would take the value for each state, V(s)I, and we will overwrite it with this expression here. In the original, this discrete value iteration algorithm was to V(s)I and we will set V(s)I to be equal to that, I think. Now we have in the continuous state case, we have an infinite continuous set of states and so you can't discretely set the value of each of these to that. So what we'll do instead is choose the parameters T so that V(s)I is as close as possible to this thing on the right hand side instead. And this is what YI turns out to be. So completely, what I'm going to do is I'm going to construct estimates of this term, and then I'm going to choose the parameters of my function approximator. I'm gonna choose my parameter as T, so that V(s)I is as close as possible to these. That's what YI is, and

specifically, what I'm going to do is I'll choose parameters data to minimize the sum of square differences between T [inaudible] plus 5SI. This thing here is just V(s)I because I'm approximating V(s)I is a linear function of 5SI and so choose the parameters data to minimize the sum of square differences. So this is last step is basically the approximation version of value iteration. What everything else above was doing was just coming up with an approximation to this term, to this thing here and which I was calling YI. And so confluently, for every state SI we want to estimate what the thing on the right hand side is and but there's an expectation here. There's an expectation over a continuous set of states, may be a very high dimensional state so I can't compute this expectation exactly. What I'll do instead is I'll use my simulator to sample a set of states from this distribution from this P substrip, SIA, from the state transition distribution of where I get to if I take the action A in the state as I, and then I'll average over that sample of states to compute this expectation. And so stepping through the algorithm just says that for each state and for each action, I'm going to sample a set of states. This S prime 1 through S prime K from that state transition distribution, still using the model, and then I'll set Q(a) to be equal to that average and so this is my estimate for R(s)I + G (this expected value for that specific action A). Then I'll take the maximum of actions A and this gives me YI, and so YI is for S for that. And finally, I'll run really run linear regression which is that last of the set [inaudible] to get V(s)I to be close to the YIs. And so this algorithm is called fitted value iteration and it actually often works quite well for continuous, for problems with anywhere from 6- to 10- to 20-dimensional state spaces if you can choose appropriate features. Can you raise a hand please if this algorithm makes sense? Some of you didn't have your hands up. Are there questions for those, yeah?

Student:
Is there a recess [inaudible] function in this setup?

Instructor (Andrew Ng):Oh, yes. An MDP comprises SA, the state transition probabilities G and R and so for continuous state spaces, S would be like R4 for the inverted pendulum or something. Actions with discretized state transitions probabilities with specifying with the model or the simulator, G is just a real number like .99 and the real function is usually a function that's given to you.

And so the reward function is some function of your 4-dimensional state, and for example, you might choose a reward function to be minus − I don't know. Just for an example of simple reward function, if we want a -1 if the pole has fallen and there it depends on you choose your reward function to be -1 if the inverted pendulum falls over and to find that its angle is greater than 30 ° or something and zero otherwise. So that would be an example of a reward function that you can choose for the inverted pendulum, but yes, assuming a reward function is given to you so that you can compute R(s)I for any state. Are there other questions?

Actually, let me try asking a question, so everything I did here assume that we have a stochastic simulator. So it turns out I can simply this algorithm if I have a deterministic simulator, but deterministic simulator is given a stated action, my next state is always exactly determined. So let me ask you, if I have a deterministic simulator, how would I change this algorithm? How would I simplify this algorithm?

Student: Lower your samples that you're drawing [inaudible].

Instructor (Andrew Ng):Right, so Justin's going right. If I have a deterministic simulator, all my samples from those would be exactly the same, and so if I have a deterministic simulator, I can set K to be equal to 1, so I don't need to draw K different samples. I really only need one sample if I have a deterministic simulator, so you can simplify this by setting K=1 if you have a deterministic simulator. Yeah?

Student:I guess I'm really confused about the, yeah, we sorta turned this [inaudible] into something that looks like linear state regression or some' you know the data transpose times something that we're used to but I guess I'm a little. I don't know really know what question to ask but like when we did this before we had like discrete states and everything. We were determined with finding this optimal policy and I guess it doesn't look like we haven't said the word policy in a while so kinda difficult.

Instructor (Andrew Ng):Okay, yeah, so [inaudible] matters back to policy but maybe I should just say a couple words so let me actually try to get at some of what maybe what you're saying. Our strategy for finding optimal policy has been to find some way to find V*, find some way to find the optimal value function and then use that to compute ?* and some of approximations of ?*. So far everything I've been doing has been focused on how to find V*. I just want to say one more word. It actually turns out that for

linear regression it's often very easy. It's often not terribly difficult to choose some resource of the features.

Choosing features for approximating the value function is often somewhat harder, so because the value of a state is how good is starting off in this state. What is my expected sum of discounted rewards? What if I start in a certain state? And so what the feature of the state have to measure is really how good is it to start in a certain state? And so for inverted pendulum you actually have that states where the poles are vertical and when a cart that's centered on your track or something, maybe better and so you can come up with features that measure the orientation of the pole and how close you are to the center of the track and so on and those will be reasonable features to use to approximate V*. Although in general it is true that choosing features, the value function approximation, is often slightly trickier than choosing good features for linear regression.

Okay and then Justin's questions of so given V*, how do you go back to actually find a policy? In the discrete case, so we have that ?*(s) is equal to all [inaudible] over A of that. So that's again, I used to write this as a sum over states [inaudible]. I'll just write this as an expectation and so then once you find the optimal value function V*, you can then find the optimal policy ?* by computing the [inaudible]. So if you're in a continuous state MDP, then you can't actually do this in advance for every single state because there's an infinite number of states and so you can't actually perform this computation in advance to every single state.

What you do instead is whenever your robot is in some specific state S is only when your system is in some specific state S like your car is at some position orientation or your inverted pendulum is in some specific position, posed in some specific angle T. It's only when your system, be it a factor or a board game or a robot, is in some specific state S that you would then go ahead and compute this augmax, so it's only when you're in some state S that you then compute this augmax, and then you execute that action A and then as a result of your action, your robot would transition to some new state and then so it'll be given that specific new state that you compute as augmax using that specific state S that you're in.

There're a few ways to do it. One way to do this is actually the same as in the inner loop of the fitted value iteration algorithm so because of an expectation of a large number of states, you'd need to sample some set of states from the simulator and then approximate this expectation using an average over your samples, so it's actually as inner loop of the value iteration algorithm. So you could do that. That's sometimes done. Sometimes it can also be a pain to have to sample a set of states to approximate those expectations every time you want to take an action in your MDP.

Couple of special cases where this can be done, one special case is if you have a deterministic simulator. If it's a deterministic simulator, so in other words, if your similar is just some function, could be a linear or a nonlinear function. If it's a deterministic simulator then the next state, $ST+1$, is just some function of your previous stated action. If that's the case then this expectation, well, then this simplifies to augmax of A of V* of F of I guess S,A because this is really saying S prime=F(s),A. I switched back and forth between notation; I hope that's okay. S to denote the current state, and S prime to deterministic state versus ST and $ST+1$ through the current state. Both of these are sorta standing notation and don't mind my switching back and forth between them. But if it's a deterministic simulator you can then just compute what the next state S prime would be for each action you might take from the current state, and then take the augmax of actions A, basically choose the action that gets you to the highest value state.

And so that's one case where you can compute the augmax and we can compute that expectation without needing to sample an average over some sample. Another very common case actually it turns out is if you have a stochastic simulator, but if your similar happens to take on a very specific form of $ST+1=F(s)T,AT+?T$ where this is galsie noise. The [inaudible] is a very common way to build simulators where you model the next state as a function of the current state and action plus some noise and so once specific example would be that sort of mini dynamical system that we talked about with linear function of the current state and action plus galsie noise. In this case, you can approximate augment over A, well.

In that case you take that expectation that you're trying to approximate. The expected value of V* of S prime, we can approximate that with V* of the expected value of S prime, and this is approximation. Expected value of a function is usually not equal to the value of an expectation but it is often a reasonable approximation and so that would be another way to approximate that expectation and so you choose the

actions according to watch we do the same formula as I wrote just now. And so this would be a way of approximating this augmax, ignoring the noise in the simulator essentially. And this often works pretty well as well just because many simulators turn out to be the form of some linear or some nonlinear function plus zero mean galsie noise, so and just that ignore the zero mean galsie noise, so that you can compute this quickly.

And just to complete about this, what that is, right, that V* F of SA, this you down rate as data transfers Fi of S prime where S prime=F of SA. Great, so this V* you would compute using the parameters data that you just learned using the fitted value iteration algorithm. Questions about this?

Student:[Inaudible] case, for real-time application is it possible to use that [inaudible], for example for [inaudible].

Instructor (Andrew Ng): Yes, in real-time applications is it possible to sample case phases use [inaudible] expectation. Computers today actually amazingly fast. I'm actually often surprised by how much you can do in real time so the helicopter we actually flying a helicopter using an algorithm different than this? I can't say. But my intuition is that you could actually do this with a helicopter. A helicopter would control at somewhere between 10hz and 50hz. You need to do this 10 times a second to 50 times a second, and that's actually plenty of time to sample 1,000 states and compute this expectation.

They're real difficult, helicopters because helicopters are mission critical, and you do something it's like fast. You can do serious damage and so maybe not for good reasons. We've actually tended to avoid tossing coins when we're in the air, so the ideal of letting our actions be some up with some random process is slightly scary and just tend not to do that. I should say that's prob'ly not a great reason because you average a large number of things here very well fine but just as a maybe overly conservative design choice, we actually don't, tend not to find anything randomized on which is prob'ly being over conservative. It's the choice we made 'cause other things are slightly safer. I think you can actually often do this.

So long as I see a model can be evaluated fast enough where you can sample 100 state transitions or 1,000 state transitions, and then do that at 10hz. They haven't said that. This is often attained which is why we often use the other approximations that don't require your drawing a large sample. Anything else? No, okay, cool. So now you know one algorithm [inaudible] reinforcement learning on continuous state spaces. Then we'll pick up with some more ideas on some even more powerful algorithms, the solving MDPs of continuous state spaces. Thanks. Let's close for today.

[End of Audio]

Duration: 77 minutes

1.19 Machine Learning Lecture 19[19]

Figure 1.19

MachineLearning-Lecture19

Instructor (Andrew Ng): All right, good morning. So just one administrative thing before we go into today's technical material. So let's see. The final projects poster session presentations will be on Wednesday the 12th of December. I guess that's one week from this Wednesday. So similar to the mid-term exam, if you physically live in the Bay Area then please come in person to do the poster session. So this means you guys, you know, so if on campus students, and SCPD students are that, so they live in the Bay Area. If, for some reason, you live in the Bay Area, but aren't able to come in in person to the poster session, please send us an e-mail as soon as possible at the usual class mailing address cs229qa@cscenter.edu; it's the one on the

---

[19]This content is available online at <http://cnx.org/content/m45947/1.3/>.

course website; let us know. And if you're an SCPD or SITS student and if you live physically outside the Bay Area then you're exempt from the poster session, I guess. The poster session itself will be held at 8:30 a.m. on this – so the registrar assigns classes a certain time slot for final exams, so this class doesn't have a written final. We use that time slot instead for the final poster presentations. So they're about, I don't know, I think there are, like, about 70 posters in this class. So in order for me to be able to see everything in a reasonable amount of time please prepare, like, about a two-minute presentation, so that as I go around the posters next Wednesday you can just briefly tell me about your work. And that's the short, and then as I promise every year, I personally read every single word of every single final project write-up, so don't worry about telling me everything, every little detail, of what you did because when you send me your final project write-up – so I promise I read every single word of every write-up, so I get all the details from there. But to go in the process presentation is actually first and foremost for you guys also to have a look at what other cool things people in the class are doing and based on the models, I've been flipping through the models, of all the very cool projects this year. So at the poster presentation too hopefully you can see what your colleagues, your classmates, have been doing and have fun doing that and also for me to just get a very brief sense of the things you've been doing. Okay?

The posters themselves, we will supply poster boards if you want them. You're welcome to buy your own poster boards, but in a few days I'll send instructions on where you can pick up a poster board. If you do get a poster board from us we do reuse them. We do recycle them from year to year, so if you do get one from us that I ask you to return it at the end of the poster session next Wednesday. So poster boards are, I don't know, maybe about this wide and about this tall, so you can print out several pieces of paper. Print out several slides and attach them to the poster board. And we'll provide easels, as well, for the poster presentation. Okay? So next Wednesday, come a little bit before 8:30 a.m. and we'll meet on the first floor of the Gates lobby. So around the first floor of the Gates lobby around Gates 104 and we'll set up poster there. Okay? Are there questions about the poster session? No? Okay, cool. Let's see.

Okay. So welcome back, and let's continue our discussion of reinforcement learning algorithms. On for today, the first thing I want to do is actually talk a little bit about debugging reinforcement learning algorithms and then I'll continue the technical discussion from last week on LQR, on linear-quadratic regulation. In particular, I want to tell you about an algorithm called the French dynamic programming, which I think is actually a very effective absolute controls lack reinforcement learning algorithm for many problems. Then we'll talk about Kalman filters and linear-quadratic Gaussian control, LGG. Let's start with debugging RL algorithms. And can I switch to the laptop display, please?

And so this was actually – what I'm about to do here is this is actually a specific example that I had done earlier in this quarter, but that I promised to do again. So remember, you know, what was it? Roughly halfway through the quarter I'd given a lecture on debugging learning algorithms, right? This idea that very often you run a learning algorithm and it, you know, maybe does roughly what you want to and maybe it doesn't do as well as you're hoping. Then what do you do next? And the talk of this idea that, you know, some of the really, really good people in machine learning, the people that really understand learning algorithms, they're really good at getting these things to work. Very often what they're really good at is at figuring out why a learning algorithm is working or is not working and that prevents them from doing things for six months that someone else may be able to just look at and say gee, there was no point collecting more training data, because your learning algorithm had high bias rather than high variance. So that six months you spent collecting more training data – I could have told you six months ago it was a waste of time. Right?

So these are sorts of things that some of the people are really good at machine learning, that they really get machine learning, are very good at. Well, just a few of my slides. These slides I won't actually talk about these, but these are exactly the same slides you saw last time. Actually, I'll just skip ahead, I guess. So last time you saw this discussion on – right. Diagnostics for whether you happen to have a bias problem, or a variance problem, or in other cases where the – your optimization algorithm is converging or whether it's the [inaudible] optimization objective and so on. And we'll talk about that again, but the one example that I, sort of, promised to show again was actually a reinforcement learning example, but at that time we hadn't talked about reinforcement learning yet, so I promised to do exactly the same example again, all right?

So let's go for the example. The motivating example was robotic control. Let's see. Write a – let's say

you want to design a controller for this helicopter. So this is a very typical way by which you might apply machine-learning algorithm or several machine-learning algorithms to a control problem, right? Which is you might first build a simulator – so control problem is you want to build a controller to make the helicopter hover in place, right? So the first thing you want to do is build a simulator of the helicopter. And this just means model of the state transition probabilities, a piece of SA of the helicopter, and you can do this by many different ways. Maybe you can try reading a helicopter textbook and building a simulator based on what's known about aerodynamics of helicopters. It actually turns out this is very hard to do. Another thing you could do is collect data and maybe fit a linear model, or maybe fit a non-linear model, to what the next stage is as a function of current state and current action. Okay? So there's different ways of estimating the state transition probabilities. So, now, you now have a simulator and I'm showing a screen shot of our simulator we have a stand fit on the upper right there. Second thing you might do is then choose a reward function. So you might choose this sort of quadratic cos function, right? So the reward for being at a state X is going to be minus the norm difference between a current state and some desired state of your one simple example of a reward function. And this, sort of, quadratic reward function is what we've been using in the last lecture in LQR control, linear-quadratic regulation control.

And finally, right? Random reinforcement learning algorithm in simulation, meaning that you use your model of the dynamics to try to maximize that final horizon sum of rewards and when you do that you get a policy out, which I'm gonna call the policy pi subscript RL to denote the policy output by the reinforcement learning algorithm. Okay? Let's say you do this and the resulting controller gets much worse performance than a human pilot that you hire to fly the helicopter for you. So how do you go about figuring out what to do next? Well, actually, you have several things you might do, right? You might try to improve the simulator, so there are exactly three steps. You want say maybe after the new model from the helicopter dynamics, but I think it's non – it is actually non-linear. Or maybe you want to collect more training data, so you can get a better estimates of the transition probabilities of the helicopter. Or maybe you want to fiddle with the features you used to model the dynamics of your helicopter, right? Other things you might do is, you might modify the reward function R if you think, you know, it's not just a quadratic function, maybe it's something else. I don't know. Or maybe you're unsatisfied with the reinforcement learning algorithm. Maybe you think, you know, the algorithm isn't quite doing the right thing. Or maybe you think you need to discritize the states more finely in order to apply your reinforcement learning algorithm. Or maybe you need to fiddle with the features you used in value function approximations. Okay? So these are three examples of things you might do and, again, quite often if you chose the wrong one to work on you can easily spend, you know – actually this one I don't want to say six months. You can easily spend a year or two working on the wrong thing. Hey, Dan, this is a favor. I'm, sort of, out of chalk could you wander around and help me get? Thanks.

So the team does three things; they'll copy the yellow box to the upper right of this slide. What can you do? So this is the, sort of, reasoning we actually go through on the helicopter project often and to decide what to work on. So let me just step for this example fairly slowly. So this, sort of, reasoning you might go through. Suppose these three assumptions hold true, right? Suppose that the helicopter simulator is accurate, so let's suppose that you built an accurate model of the dynamics. And suppose that, sort of, I turned two under slides, so suppose that the reinforcement learning algorithm correctly controls the helicopter in simulation. So it's a maximized that expected payoff, right? And suppose that maximizing the expected payoff corresponds to autonomous flight, right? If all of these three assumptions holds true, then that means that you would expect the learned controller pi subscript RL to fly well on the actual helicopter. Okay? So this is the – I'm, sort of, showing you the source of the reasoning that I go through when I'm trying to come up with a set of diagnostics for this problem. So these are some of the diagnostics we actually use routinely on various revised control problems. So pi subscript RL, right? We said it doesn't fly well on the actual helicopter. So the first diagnostic you want to run is just check if it flies well in simulation, all right? So if it flies well in simulation, but not in real life, then the problem is in the simulator, right? Because the simulator predicts that your controller, the pi subscript RL, flies well, but it doesn't actually fly well in real life. So if this holds true, then that suggests a problem is in the simulator. Question?

Student:[Inaudible] the helicopter pilots try to fly on the simulator? Do you have the real helicopter

pilots try to fly on the simulator?

Instructor (Andrew Ng): Do I try to have the real helicopter flying simulator?

Student: Real helicopter pilots.

Instructor (Andrew Ng): Oh, I see. Do we ask the helicopter pilots to fly in simulation? So, yeah. It turns out one of the later diagnostics you could do that. On our project we don't do that very often. We informally ask the pilot, who's one of the best pilots in the world, Gary Zoco, to look at the simulator sometimes. We don't very often ask him to fly in the simulator. That answers your question. But let me actually go on and show you some of the other diagnostics that you might use then. Okay? Second is let me use pi subscript human to denote the human control policy, right? This is pi subscript human is policy that, you know, however the human flies it. So one thing to do is look at the value of pi RL compared to the value of pi subscript human. Okay? So what this means really is, look at how the helicopter looks like when it's flying under control of the pi subscript RL and look at what the helicopter does when it's flying under the human pilot control and evaluate – and then, sort of, compute the sum of rewards, right? For your human pilot performance and compute the sum of rewards for the learning controller performance and see on, say, the real helicopter or that question or you can do this on the real helicopter or on simulation actually, but you can see does the human obtain a higher or a lower sum of rewards on average than does the controller you just learned. Okay? And then the way you do this you actually can go and fly the helicopter and just measure the sum of rewards, right? On the actual sequence of states the helicopter flew through, right? So if this condition holds true, right? Where my mouse pointer is if – oh, excuse me. Okay. If this condition holds true where my mouse pointer is, if it holds true that pi subscript RL is less than pi subscript human, those of you watching online I don't know if you can see this, but this V pi subscript RL of as zero less than V pi subscript human of as zero. But if this holds true, then that suggests that a problem is in the reinforcement learning algorithm because the human has found a policy that obtains a higher reward than does the reinforcement learning algorithm. So this proves, or this shows, that your reinforcement learning algorithm is not maximizing the sum of rewards, right? And lastly, the last condition is this – the last test is this, if the inequality holds in the opposite direction – so if the reinforcement learning algorithm obtains a higher sum of rewards on average than does the human, but the reinforcement learning algorithm still flies worse than the human does, right? Then this suggests that the problem is in the cos functions than the reward function because the reinforcement learning algorithm is obtaining a higher sum of rewards than the human, but it still flies worse than the human. So that suggests that maximizing the sum of rewards does not correspond to very good autonomous flight. So if this holds true, then the problem is in your reward function. Or in other words, the problem is in your optimization objective and then rather than the algorithm that's trying to maximize the optimization objective and, so you might change your optimization objective. In other words, you might change reward function. Okay?

So, of course, this is just one example of how you might debug a reinforcement learning algorithm. And these particular set of diagnostics happen to apply only because we're fortunate enough to be able to an amazingly good human pilot who can fly a helicopter for us, right? If you didn't have a very good pilot then maybe some of these diagnostics won't apply and you'll have to come up with other ones. But want to go through this again as an example of the source of diagnostics you use to debug a reinforcement learning algorithm. And the point of this example isn't so much that I want you to remember the specifics of the diagnostics, right? The point of this is really for your own problem, be it supervised learning, unsupervised learning, reinforcement learning, whatever. You very often have to come up with your own diagnostics, your own debugging tools, to figure out why an algorithm is working and why an algorithm isn't working. And this is one example of what we actually do on the helicopter. Okay? Questions about this? Yeah, Justin?

Student:I'm just curious how do you collect, like, training data? Like in our homework the pendulum fell over lots of times, but how do you work that with an expensive helicopter?

Instructor (Andrew Ng): Yeah. So, I see, right. So on the helicopter the way we collect data to learn [inaudible] and improbabilities is we usually – not – done lots of things, but to first approximation, the most basic merger is if you ask a human pilot to just fly the helicopter around for you and he's not gonna crash it. So you can collect lots of data as is being controlled by a human pilot. And, as I say, it turns out in data collection the, sort of, a few standard ways to collect data are to let you – so a helicopter can do lots of

things and you don't want to collect data representing only one small part of the flight regime. So concretely what we often do is ask the pilot to carry out frequency sweeps and what that means, very informally, is you imagine them holding a control stick, right? In their hand. Frequency sweeps are a process where you start off making very slow oscillations and then you start taking your control stick and moving it back and forth faster and faster, so you sort of sweep out the range of frequencies ranging from very slow slightly [inaudible] by oscillations until you go faster and faster and you're sort of directing the control seat back and forth. So this is – oh, good. Thank you. So that's, sort of, one standard way that we use to collect data on various robotics. It may or may not apply to different robots or to different systems you work on. And, as I say, in reality we do a lot of things. Sometimes we have a controllers collect data autonomously too, but that's other more complex algorithms. Anything else?

Student: In the first point the goal communicates that the other is perhaps not mapping the actions similar to the simulator, so in [inaudible] this could be very common that you could have [inaudible]? Just any specific matter that pilots use to take care of that variable, so all [inaudible]?

Instructor (Andrew Ng): Yeah, right. So you're saying like point one may not be simulated accurate; it may be that the hardware is doing something strange with the control. There is concluding the controls through some action and through some strange transformation because it's actually simulating it as a helicopter. I don't know. Yeah. I've definitely seen that happen on some other robots before. So maybe diagnostic one here is a better form as deciding whether the simulator matches the actual hardware, I don't know. Yeah. That's another class of those to watch out for. If you suspect that's the case, I can't think of a good, sort of, diagnostic right now to confirm that, but that, again, before – that would be a good thing to try to come up with a diagnostic for to see if there might be something wrong with the hardware. And I think these are by no means the definitive diagnostics or anything like that. It's just sort of an example, but it would be great if you come up with other diagnostics to check that the hardware is working properly that would be a great thing to do, too. Okay. Is this – okay, last couple of questions before we move on?

Student: You said the reward function was?

Instructor (Andrew Ng): Oh, in this example, what I was just using a quadratic constant. On the helicopter we often use things that are much more complicated.

Student:[Inaudible] You have no way of knowing what is the, like, desired focus?

Instructor (Andrew Ng): Yeah. So you can, sort of, figure out where – ask the human pilot to hover in place and guess what his desire was. Again, these aren't constants telling you to, yeah.

Student:[Inaudible] physics that are based learning problems. Do you – does it actually work best to use a physical finian model? You know, just, sort of, physics tell or you just sort of do learning on [inaudible]?

Instructor (Andrew Ng): Yeah. The physics models work well, right? So the answer is it varies a lot from problem to problem. It turns out that the aerodynamics of helicopters, I think, aren't understood well enough that you can look at the "specs" of a helicopter and build a very good physics simulator. So on the helicopter, we actually learn the dynamics. And I don't know how to build a physics model. For other problems, like if you actually have an inverted pendulum problem or something, there are many other problems for which the aerodynamics are much better understood and for which physic simulators work perfectly fine, but it depends a lot – on some problems physic simulators work great on some they probably aren't great on all. Okay? Cool.

So, I guess, retract the chalkboard, please. All right. So how much time do I have? Twenty minutes. Okay. So let's go back to our discussion on LQR Control, linear-quadratic regulation control, and then I want to take that a little bit further and tell you about the, sort of, one variation on the LQR called differential dynamic programming. Just to recap, just to remind you of what LQR, or linear-quadratic regulation control, is, in the last lecture I defined a horizon problem where your goal is to maximize, right? Just sort of find the horizon sum of rewards, and there's no discounting anymore, and then we came up with this dynamic programming algorithm, right? Okay.

Then we came up with this dynamic programming algorithm in the last lecture, where you compute V star of capital T that's one value function for the last time step. So, in other words, what's the value if your star in the state S and you just get to take one action and then the clock runs out. The aerodynamic programming algorithm that we repeatedly compute V star lowercase t in terms of V star t plus one, so we

compute V star capital T and then recurs backwards, right? And so on, until we get down to V star zero and then pi star was given by, as usual, the augmax of the thing we had in the definition of the value function.

So last time, the specific example we saw of – or one specific example, that sort of define a horizon problem that we solved by DP was the LQR problem where we worked directly with continuous state and actions. And in the LQR problem we had these linear dynamics where the state ST plus one is a linear function of the previous state and action and then plus this Gaussian noise WT, which is covariance sigma W. And I said briefly last time, that one specific way to come up with these linear dynamics, right? Oh, excuse me. One specific way to take a system and come up with a linear model for it is if you have some simulator, say, right? So in this cartoon, the vertical axis represents ST plus one and the horizontal axis represents STAT. So say you have a simulator, right? And let's say it determines [inaudible] simulator, so we have a [inaudible] simulator that tells you what the next state is ST plus one is a function of previous state and action. And then you can choose a point around which to linearize this simulator, by which I mean that you choose a point, an approximate your – approximate the function F using a linear function, this tangent, to the function F at that point. So if you do that you have ST plus one equals – shoot.

Sorry about writing down so many lines, but this will be the linearization approximation to the function F where I've taken a linearization around the specific point as bar T, A bar T. Okay? And so you can take those and just narrow it down to a linear equation like that where the next state ST plus one is now some linear function of the previous state ST and AT and these matrixes, AT and BT, will depend on your choice of location around which to linearize this function. Okay? I said last time, that this linearization approximation you, sort of, expect to be particularly good in the vicinity of, as bar T, A bar T because this linear function is a pretty good approximation to F, right? So if in this little neighborhood there. And – yes?

Student:[Inaudible] is there an assumption that you are looking at something the second recently indicates like the helicopter, are you assuming pilot behavior is the same as [inaudible] behavior or –

Instructor (Andrew Ng):Yeah, right. So let me not call this as an assumption. Let me just say that when I use this algorithm, when I choose to linearize this way, then my approximation would be physically good in the vicinity here and it may be less good elsewhere and, so let me – when I actually talk about DDP I actually make use of this property. Which is why I'm going over it now. Okay? But, yeah. There is an intuition that you want to linearize near the vicinity—near of states, so you expect your system to spend the most time. Right.

So, okay. So this is how you might come up with a linear model and if you do that then, oh, you can – let's see. So for LQR we also have this sort of quadratic reward function, right? Where the matrixes UT and VT are positive semi-definite, so the rewards are always negative, that's this minus sign. And then if you take exactly the dynamic programming algorithm, that I've written down just now, then – lets see. It turns out that the value function at every state, excuse me. It turns out the value function for every time step will be a quadratic function of the state and can be written like that. And so you initialize the dynamic programming algorithm as follows. And I just write this down, but there's actually just one property I want to point out later, but this equation is, well, somewhat big and hairy, but don't worry about most of its details.

Let me just put this. Shoot, there's one more equation I want to fit in. Well, okay. All right. So it turns out the value function is a quadratic function where V star T is this and, so you initialize the dynamic programming step with this. This fi and this si gives you V capital T and then it records backwards. So these two equations express – will give you V subscript T as a function of VT plus one. Okay? So it incurs backwards for this learning algorithm. And the last thing, get this on the same board, so – sorry about the disorganized use of the boards. I just wanted this on the same place.

And, finally, the actual policy pi star of ST is given by some linear function of ST, so LT here is a matrix where LT is equal to – numerous times. Okay? And so when you do this you now have the actual policy, right? So just concretely, you run the dynamic programming algorithm to compute fi T and si T for all values of T and then you plug it in to compute the matrixes LT and now you know the optimal action stake and [inaudible]. Okay? So there's one very interesting property – these equations are a huge mess, but you can re-derive them yourself, but don't worry about the details. But this is one specific property of this

dynamic programming algorithm that's very interesting that I want to point out. Which is the following. Notice that to compute the optimal policy I need to compute these matrixes LT and notice that LT depends on A, it depends on B, it depends on D, and it depends on fi, but it doesn't depend on si, right? Notice this further that when I carry out my dynamic programming algorithm my recursive definition for si – well it depends on – oh, excuse me. It should be si T, right. Si T plus one. Okay. In order to carry out my dynamic programming algorithm, right? For si T I need to know what si T plus one is. So si T depends on these things, but in order to carry out the dynamic programming for fi T, fi T doesn't actually depend on si T plus one, right? And so in other words in order to compute the fi T's I don't need these si's. So if all I want is the fi's I can actually omit this step of the dynamic programming algorithm and not bother to carry out the dynamic programming algorithm in terms of the fi's. And then having done my dynamic programming algorithm just for – excuse me, I misspoke. You – I can forget about the si's and just do the dynamic programming updates for the fi matrixes and then having done my DP updates for the fi T I can then plug this into this formula to compute LT. Okay?

And so one other thing about it is you can, to be slightly more efficient – efficiency isn't really the issue, but if you want you can actually forget about the fi T's. You actually don't need to compute that at all. Now, the other interesting property of this is that the matrix sigma W appears only in my DV update for the si T's. It doesn't actually appear in my updates for the fi T's. So you remember – well, my model was that ST plus one equals ATST plus VT, AT plus WT where these noise terms, WT, had a covariance sigma W and so the only place that appears – the covariance of the noise terms of appears is in those IT's, but I just said that they can do this entire [inaudible] ordering algorithm without the si T's. So what this means is that you can actually find the optimal policy without knowing what the covariance of the noise terms are. Okay?

So this is a very special property of LQR systems and once you change anything, once you go away from a linear dynamical system, or once you change almost any aspect of this because at discreet states or discreet actions or whatever and once you change almost any aspect of this problem this property will no longer hold true because this is a very special property of LQR systems that the optimal policy does not actually depend on the noise magnitude of these noise terms. Okay? And the only important property is that the noise function of zero mean. So there's this intuition that to compute the optimal policy you can just ignore the noise terms. Or as if, so as long as you know the expected value of your state ST plus one – write down. On average ST plus one is ATST plus BTAT, then there's as if you can ignore the noise in your next state ST plus one. And the optimal policy doesn't change. Okay?

So we'll actually come back to this in a minute. Later on we'll talk about Kalman filters. We'll actually use this property of LQR systems. Just to point out, note that the value function does depend on the noise covariance, right? The value function here does depend on si T. So the larger the noise in your system the worse your value function. So this does depend on the noise, but it's the optimal policy that doesn't depend on the noise. We'll use this property later. Okay. So let's see how we're doing on time. Let's see. Right. Okay. So let's put this aside for now. What I want to do now is tell you about one specific way of applying LQR that's called differential dynamic programming. And as in most of the example, think of try to control a system, like a helicopter or a car or even a chemical plant, with some continuous state. So for the sake of thinking through this example, just imagine trying to control a helicopter. And let's say you have some simulator that espece with the next state is a function of the previous data in action, right? And for this let's say the model of your simulator is non-linear, but and determinalistic. Okay? So I say just now, that the noise terms don't matter very much. So let's just work with the term simulator for now, but let's say F is non-linear. And let's say there's some specific trajectory that you want the helicopter to follow, all right? So I want to talk about how to apply LQR to get helicopter or a car or a chemical plant where your state variables may depend on the amounts of different chemicals and the mixes of chemicals you have in different batch, really. It's really easy to think about a helicopter. Let's say there's some trajectory you want the helicopter to follow. So here's what the differential dynamic programming it does. First step is come up with what I'm gonna call some nominal trajectory, right? And so we're gonna call this S zero A zero. Okay? So one way to come up with this would be if you had some very bad controller – someone hacked a controller for flying a helicopter is not a good controller at all. But you might then go ahead and fly the helicopter

using a very bad, a very sloppy, controller and you get out some sequence of states and actions, right? So I'm gonna – and I just call this sequence of states and actions the trajectory – the nominal trajectory. Then I will linearize F around this normal trajectory. Okay? So i.e., right? I'll use that same thing. So for times si T our approximate ST plus one, as this linearization thing that we just saw, times – plus the other term. Okay? And then you distill this down to sum ATST plus BTST. Okay? So this will actually be the first time that I'll make explicit use of the ability of LQR or these finer horizon problems to handle non-stationery dynamics. In particular, for this example, it will be important that AT and BT depend on time – oh, excuse me. Okay?

So the intuition is that even if this is a pretty sloppy controller, or even if you had a pretty bad controller come up with your original normal trajectory, you still expect maybe, right? You'd expect your state and action at time T to be maybe reasonably similar to what even the sloppy controller had done, right? So you want a fly trajectory maybe you want to make a 90-degree turn. Maybe if a bad controller that does a pretty sloppy job, but at any given in time you're still moving around this trajectory. So this is really telling you where along, say, the 90-degree turn trajectory, just very roughly, where along the trajectory you expect to be at any given time and so let's linearize around that point. Okay?

Then you would – having found the linear model you run LQR to get the optimal policy for this specific linear model and now you have a better policy. And the final thing you do is – boy, I'll write this on a different board, I guess. Okay. Shoot. The last step is you use a simulator, a model, to come up with a new normal trajectory. So i.e., okay? So now you take the controller you just learned and basically try flying your helicopter in your simulator. So you initialize the simulator to the initial state, and I'll call the S bar zero, and you'll run every time step. You choose an action which I'll call A bar T, using the controller pi T that you just learned using LQR. And then you simulate forward in time, right? You use the simulator, the function F, to tell you what the next state S bar T plus one will be when your previous state and action is bar T A bar T. And then you linearize around this new trajectory and repeat. Okay? So now you have a new normal trajectory and you linearize your simulator around this new trajectory and then you repeat the whole procedure. I guess going back to step two of the algorithm. And this turns out to be a surprisingly effective procedure. So the cartoon of what this algorithm may do is as follows. Let's say you want to make a 90-degree turn on the helicopter let's see one, you know, a helicopter to follow a trajectory like that. Follow up of a very bad controller, I just, you know, hack up some controller, whatever. Have some way to come up with an initial normal trajectory. Maybe your initial controller overshoots the turn, takes the turn wide, right? But now you can use these points to linearize the simulator. So linearize in a very non-linear simulator and the idea is that maybe this state isn't such a bad approximation. That maybe a linearization approximation at this sequence of states will actually be reasonable because your helicopter won't exactly be on the states, but will be close to the sequence of states of every time step. So after one duration of DDP, that's the target trajectory, maybe you get a little bit closer and now you have an even better place around to linearize. Then after another linearization of DDP you get closer and closer to finding exactly the trajectory you want. Okay?

So turns out DDP is a sort of – it turns out to be a form of a local search algorithm in which you – on each iteration you find a slightly better place to linearize. So you end up with a slightly better control and you repeat. And we actually do this – this is actually one of the things we do on the helicopter. And this works very well on many – this works surprisingly well – this works very well on many problems. Cool. I think – I was actually going to show some helicopter videos, but in the interest of time, let me just defer that to the next lecture. I'll show you a bunch of cool helicopter things in the next lecture, but let me just check if there are questions about this before I move on. Yeah?

Student:[Inaudible]

Instructor (Andrew Ng):In this sample? Yes, yeah, right, yeah. So I'm going to – let's pick some kind of horizon T. So I'm going to run through my entire trajectory in my simulator, so I end up with a new nominal trajectory to linearize around, right? Okay? Yeah?

Student: So does this method give you, like, a Gaussian for performing, like, a certain action? Like you talked about, like, the 90-degree turn thing or something.

Instructor (Andrew Ng):Right.

Student: So is this from one, like, is this from one, like, one 90-degree turn or can you [inaudible]?

Instructor (Andrew Ng): Yeah. So it turns out what − so this is used clear − let's see. Go and think about this as if there's a specific trajectory that you want to follow. I'm just gonna, car or helicopter or it could be in a chemical plant, right? If there's some specific sequence of states you expect the system to go through over time, so that you actually want to linearize at different times − excuse me. So, therefore, the different times you want different linear approximations, your dynamics, right? So I actually start to laugh over stationary simulator, right? I mean, this function F, it may be the same function F for all time steps, but the point of DDP is that I may want to use different linearizations for different time steps. So a lot of the inner loop of the algorithm is just coming up with better and better places around to linearize. Where at different times I'll linearize around different points. Does that make sense? Cool. Okay, cool. So that was DDP.

And I'll show examples of DDP results in the next lecture. So the last thing I wanted to do was talk about Kalman filters and LQG control, linear-quadratic Gaussian control. And what I want to do is actually talk about a different type of MDP problem where we don't get to observe the state explicitly, right? So far in every one I've been talking about, I've been assuming that every time step you know what the state of the system is, so you can compute a policy to some function of the state is in. If you've all ready had that, you know, the action we take is LT times ST, right? So to compute the action you need to know what the state is.

What I want to do now is talk about the different type of problem where you don't get to observe the state explicitly. The fact − before we even talk about the control let me just talk about the different problem where − just forget about control for now and just look at some dynamical systems where you may not get to observe the state explicitly and then only later we'll tie this back to controlling some systems. Okay? As a concrete example, let's say as, sort of, just an example to think about, imagine using a radar to track the helicopter, right? So we may model a helicopter, and this will be an amazingly simplified model of a helicopter, as, you know, some linear dynamical systems. So [inaudible] ST plus one equals AST plus WT, and we'll forget about controls for now, okay? We'll fill the controls back in. And just with this example, I'm gonna use an extremely simplified state, right? Where my state is just a position in velocity in the X and Y directions, so you may choose an A matrix like this as a − okay?

As an extremely simple − as a, sort of, an extremely simplified model of what the dynamics of, like, a plane or object moving in 2-D may look like. So just imagine that you have simulation and you have a radar and you're tracking blips on your radar and you want to estimate the position, or the state, of the helicopter as just as its XY position and its XY velocity and you have a very simple dynamical model of what the helicopter may do. So this matrix, this just says that XT plus one equals XT plus X star T plus noise, so that's this first equation. The second equation says that X star T plus one equals 0.9 times X star T plus nine. Yes, this is an amazingly simplified model of what a flying vehicle may look like.

Here's the more interesting part, which is that with − if you're tracking a helicopter with some sensor you won't get to observe the full state explicitly. But just for this cartoon example, let's say that we get to observe YT, which is CST plus VT where the VT is a random variables − Gaussian random variables with, say, zero mean and a Gaussian noise with covariance given by sigma V. Okay? So in this example let's say that C is that and − so CST is equal to XY, right? Take this state vector and multiply it by Z, you just get XY. So let's see what the sensor, maybe a radar, maybe a vision system, I don't know, something that only gets to observe the position of the helicopter that you're trying to track.

So here's the cartoon. So a helicopter may fly through some sequence of states, let's say it flies through some smooth trajectory, whatever. It makes a slow turn. So the true state is four-dimensional, but I'm just drawing two dimensions, right? So maybe you have a camera sensor down here, or a radar or whatever, and for this cartoon example, let's say the noise in your observations is larger in the vertical axis than the horizontal axis. So what you get is actually one sample from the sequence of five Gaussians. So you may observe the helicopter there at times step one, observe it there at time step two, observe it there at time three, time four, time five. Okay? All right. So that's what your − there's a sequence of positions that your camera estimate gives you. And given these sorts of observations, can you estimate the actual state of the system? Okay? So these orange things, I guess, right? Okay?

These orange things are your observations YT. And test for the state of helicopter every time. Just for it, so the position of the helicopter at every time. Clearly you don't want to just rely on the orange crosses because that's too noisy and they also don't give you velocities, right? So you only observe the subset of the state of variables. So what can you do? So concretely – well, you don't actually ever get to observe the true positions, right? All you get to do is observe those orange crosses. I guess I should erase the ellipses if I can. Right. You get the idea. The question is given – yeah. You know what I'm trying to do. Given just the orange crosses can you get a good estimate of the state of the system at every time step?

So it turns out that – well, so what you want to do is to estimate the distribution on the state given all the previous observations, right? So given observations, you know, one, two, three, four, and five, where is the helicopter currently? So it turns out that the random variables, S zero, S one, up to ST and Y1 are to ST, have a joint Gaussian distribution, right? So one thing you could do is construct a joint Gaussian distribution – can define vector value random variable Z, S zero, S one, up to ST, Y1 up to YT, right? So it turns out that Z will have some Gaussian distribution with some mean and some covariance matrix. Using the Gaussian marginalization and conditioning formulas. But I think way back when we talked about factor analysis in this class, we talked about how to compute marginal distributions and conditional distributions of Gaussians. But using those formulas you can actually compute this thing. You can compute, right? You can compute that conditional distribution. This will give a good estimate of the current state ST. Okay?

But clearly this is a extremely computationally inefficient way to do so because these means and covariance matrixes will grow linearly with the number of time steps as you're tracking a helicopter over tens of thousands of time steps. They were huge covariance matrixes, so this is a conceptually correct way, but just a computational not reasonable way to perform this computation. So, instead, there's an algorithm called the Kalman filter that allows you to organize your computations efficiently and do this. Just on the side, if you remember Dan's discussion section on HMM's the Kalman filter model turns out to actually be a hidden Markov model. These Kalman's are only for those of you that attended Dan's discussion section. If not then what I'm about to say may not make sense. But if you remember Dan's kind of section of the hidden Markov model, it actually turns out that the Kalman filter model, this linear dynamical system with observations is actually an HMM problem where – let's see. Unfortunately, the notation's a bit different because Dan was drawing from, sort of, a clash of multiple research communities using these same ideas. So the notation that Dan used, I think, was developed in a different community that clashes a bit with the reinforcement learning community notations. So in Dan's notation in the HMM section, Z and X were used to denote the state and the observations. Today, I'm using S and X to denote the state and the observations. Okay?

But it turns out what I'm about to do turns out to be a hidden Markov model with continuous states rather than discrete states, which is under the discretion section. Okay. If you didn't attend that discussion section then forget everything I just said in the last minute. So here's the outline of the Kalman filter. It turns out that, so it's a cursive algorithm. So it turns out that if I have computed P of ST given Y1 up to YT, the Kalman filter organizes these computations into steps. The first step is called the predict step. Where given P of ST – where you all ready have P of ST given Y1 up to YT and you compute what P of ST plus one given Y1 up to YT is. And then the other step is called the update step. Where given this second line you compute this third line. Okay? Where having taken account only observations of the time T you know incorporate the lots of the observations up to time T plus one.

So concretely – oh, and let's see. In the predict step it turns out that – so what I'm going to do is actually just outline the main steps of the Kalman filter. I won't actually derive the algorithm and prove it's correct. It turns out that, I don't know, working out the actual proof of what I'm about to derive is probably significantly – it's probably, I don't know, about as hard, or maybe slightly easier, than many of the homework's you've done all ready. So and you've done some pretty amazingly hard homework, so you can work out the proof for yourself. It's just write out the main outlines and the conclusion of the algorithm. So for the acceptance of the vest T given Y1 after YT. If that is given by that then where – okay? So given ST, having computed the distribution ST given Y1 through YT – and computed the distribution of ST plus one given Y1 through YT as Gaussian, with this mean and this covariance, where you compute the mean and covariance using these two formulas. And just as a point of notation, right? I'm using ST and YT to denote the true states in observations. So the ST is the unknown true state. Okay? ST is whatever state

this one is in and you actually don't know what ST is because you don't get to observe this.

And, in contrast, I'm using these things like ST given T, ST plus one given T, sigma T given T, and so on. These things are the results of your computations, right? So these things are actually things you compute. So I hope the notations are okay, but these – ST is the unknown true state, right? Whereas these things, ST equals one given T and so on, these are things that you compute inside your algorithm. Okay? So that was the predict step.

And in the update step, you find that – well, okay? And so that's the updates of the Kalman filter where you compute this in terms of your ST given Y1 through YT. So after having performed the most recent Kalman filter update you find that, right? Your perceived distribution on the estimate of ST plus one, given all your observations so far, is that it's Gaussian with mean given by this and variance given by that. So, informally, this thing ST plus one given T plus one is our best estimate for ST plus one, right? Given all the observations we've had up to that time. Okay?

And, again, the correctness of these equations – the fact that I'm actually computing this mean and covariance of this conditional Gaussian distribution, you can – I'll leave you to sort of prove that at home if you want. Okay? I'm actually gonna put this together with LQR control in a second, but, so before I do that let me check if you've got questions about this? Actually let me erase the board while you take a look at that. Right. Okay. Any questions for Kalman filters? Yeah?

Student: How is it computationally less intensive than compute some drawing Gaussian distribution and then find the conditional –

Instructor (Andrew Ng): Very quickly. Yeah, of course. So how is this less computationally intensive than the original method I talked about, right? So in the original method I talked about – wow, this is really back and forth. I said, let's construct a Z, which was this huge Gaussian thing, right? And figure out what the mean and covariance matrix of Z is. So sigma will be like R – it'll be – well, it'll be roughly, right? A T by T matrix, right? This is actually – or the T by T is actually T times number of state variables plus number of observation variables by that. This is a huge matrix and as the number of times it increases sigma will become bigger and bigger. So the conditional and marginalization operations require things like computing the inverse of T or subsets of T. So the naïve way of doing this will cos on the order of TQ computation, if you do things naively, right? If – because inverting like a T by T matrix cos on the order of TQ, roughly.

In contrast, the Kalman filter algorithm, like I said, over here. I just have the update step. On the other board I had the predict step. But you can carry out the computation on both of these lines and it's actually constant time. So on every time step you perform these Kalman filter updates. So if every time you get one more observation you perform one more Kalman filter update and the computation of that doesn't depend on it's – or the one time for every time step. So the amount of stuff you need to keep around in memory doesn't grow linearly with the number of time steps you see. Okay? Because – actually what – I think I just realized why – so, yes. Actually this is the way we actually run Kalman filters, which is initially I have just my first observation. So I then compute P of X1 given Y1, right? And now I know why I think my helicopter is at time step one. Having computed this there may be some time passes, like a second passes, and then I get another observation and what I'll do is I'll combine these two together to get P of X2 given Y1 and Y2, right? And then may be another second passes in time and I get another observation. So my helicopters move a little bit more, because another second's passed and I get another observation. What I do is I combine these two to compute P of SV given Y1, Y2, Y3. And it turns out that in order to compute this I don't need to remember any of these earlier observations. Okay? So this is how you actually run it in real time say. Okay? Cool. So – oh, drat, running out of time. The last thing I want to do is actually put these things together. So putting it together – putting Kalman filters together with LQR control you get an algorithm called LQG control, which stands for linear-quadratic Gaussian. But in this type of control problem, we have a linear dynamical system. So I'm now adding actions back in, right? So now B times AT. Okay? And then, so LQG problem, or linear-quadratic Gaussian problem, I have a linear dynamical system that I want to control and I don't get to observe the states directly. I only get to observe these variables YT. Okay? So I only get noisy observations of the actual state. So it turns out that you can solve an LGG control problem as follows. At every time step, we'll use a Kalman filter to estimate the state, right? So concretely – let's say you know the initial state. Then you initialize this to be like that. If you know that

the initial state is some state as zero, you initialize that as zero and that or, whatever, right? And this is just – well, okay? If you don't know the initial state exactly, then this is just a mean of your initial state estimate and that would be your covariance or your initial state estimate. So just initialize your Kalman filter this way. And then you use the Kalman filter on every step to estimate what the state is. So here's the predict step, right? Previously we had ST plus one give T equals – and so on. So this is your predict step and then you have an update step, same as before. The one change I'm gonna make to the predict step is now I'm going to take this into account as well. This is just saying suppose my previous state was ST given T, what do I think my next state ST plus one given T will be given no other observations and the answer is, you know, it's really just this equation, AST given T plus BAT.

And then, so this takes care of, sort of, the observations. And then the other thing you do is compute LT's using LQR, right? Assuming – then the other thing you do is you just look at the linear dynamical systems, and forget about the observations for now, and compute the optimal policy – oh, right. Previously we had that you would choose actions AT equals to LT times ST, right? So the optimal policy we said was these matrixes, LT times ST. So the other part of this problem you would use LQR to compute these matrixes LT, ignoring the fact that you don't actually observe the state. And the very final step of LQR control is that – well, when you're actually flying a helicopter, when you're actually doing whatever you're doing, you can't actually plug in the actual state because in LGG problem you don't get to observe the state exactly. So what you do when you actually execute the policy is you choose the action according to your best estimate of the state. Okay?

So in other words, you don't know what ST is, but your best estimate of the state at any time is this S of T given T. So you just plug those in and take LT times your best estimate of the state and then you go ahead and execute the action AT on your system, on your helicopter, or whatever. Okay? And it turns out that for this specific class of problems, this is actually optimal procedure. This will actually cause you to act optimally in your LQG problem. And there's this intuition that, earlier I said, in LQR problems it's almost as if the noise doesn't matter and in a pure LQR problem the WT terms don't matter. It's as if you can ignore the noise. So it turns out that by elaborating that proof, which I'm not gonna do you can – you're welcome to proof for yourself at home. It's that intuition means that you can actually ignore the noise in your observations as well. The ST given T is some of your best estimate. So it's as if your true state ST is equal to ST given T plus noise. So in LQG control, what we're going to do is ignore the noise and just plug in this ST given T and this turns out the optimal thing to do. I should say, this turns out to be a very special case of a problem where you can ignore the noise and still act optimally and this property – this actually is something called the separation principle where you can design an algorithm for estimate the states and design an algorithm for controlling your system. So just glom the two together and that turns out to be optimal. This is a very unusual property and it pretty much was true only for LQG. It doesn't hold true for many systems. Once you change anything, one's that's non-linear, you know, some other noise model of one that's non-linear once this – I don't know. Once you change almost anything in this problem this will no longer hold true. The – and just estimate the states and plug that into a controller that was designed, assuming you could observe the states fully. But that once you change almost anything this will no longer turn out to be optimal. But for the LQG problem specifically, it's kind of convenient that you can do this. Just one quick question to actually close

Student:[Inaudible]

Instructor (Andrew Ng): Oh, yes. Yeah. In every embassy wing – in everything I've described I'm assuming that you're already learned A and B or something, so to –

Student:[Inaudible]

Instructor (Andrew Ng): Yeah, right. Okay. Sorry we're running a little bit late; let's close for today and next time I'll talk a bit more about these partially observed problems.

[End of Audio]

Duration: 79 minutes

1.20 Machine Learning Lecture 20[20]

Figure 1.20

Instructor (Andrew Ng):Okay. Good morning. Just one quick announcement before I start. Poster session, next Wednesday, 8:30 as you already know, and poster boards will be made available soon, so the poster boards we have are 20 inches by 30 inches in case you want to start designing your posters. That's 20 inches by 30 inches. And they will be available this Friday, and you can pick them up from Nicki Salgudo who's in Gates 187, so starting this Friday. I'll send out this information by e-mail as well, in case you don't want to write it down.

For those you that are SCPD students, if you want to show up here only on Wednesday for the poster session itself, we'll also have blank posters there, or you're also welcome to buy your own poster boards. If you do take poster boards from us then please treat them well. For the sake of the environment, we do ask you to give them back at the end of the poster session. We'll recycle them from year to year. So if you do take one from us, please don't cut holes in it or anything. So welcome to the last lecture of this course. What I want to do today is tell you about one final class of reinforcement learning algorithms. I just want to say a little bit about POMDPs, the partially observable MDPs, and then the main technical topic for today will be policy search algorithms. I'll talk about two specific algorithms, essentially called reinforced and called Pegasus, and then we'll wrap up the class. So if you recall from the last lecture, I actually started to talk about one specific example of a POMDP, which was this sort of linear dynamical system. This is sort of LQR, linear quadratic revelation problem, but I changed it and said what if we only have observations YT. And what if we couldn't observe the state of the system directly, but had to choose an action only based on some noisy observations that maybe some function of the state?

So our strategy last time was that we said that in the fully observable case, we could choose actions – AT equals two, that matrix LT times ST. So LT was this matrix of parameters that [inaudible] describe the dynamic programming algorithm for finite horizon MDPs in the LQR problem. And so we said if only we knew what the state was, we choose actions according to some matrix LT times the state. And then I said in the partially observable case, we would compute these estimates. I wrote them as S of T given T, which were our best estimate for what the state is given all the observations. And in particular, I'm gonna talk about a Kalman filter which we worked out that our posterior distribution of what the state is given all the observations up to a certain time that was this.

So this is from last time. So that given the observations Y one through YT, our posterior distribution of the current state ST was Gaussian would mean ST given T sigma T given T. So I said we use a Kalman filter to compute this thing, this ST given T, which is going to be our best guess for what the state is currently. And then we choose actions using our estimate for what the state is, rather than using the true state because we don't know the true state anymore in this POMDP. So it turns out that this specific strategy actually allows you to choose optimal actions, allows you to choose actions as well as you possibly can given that this is a POMDP, and given there are these noisy observations. It turns out that in general finding optimal policies with POMDPs – finding optimal policies for these sorts of partially observable MDPs is an NP-hard problem. Just to be concrete about the formalism of the POMDP – I should just write it here – a POMDP formally is a tuple like that where the changes are the set Y is the set of possible observations, and this O subscript S are the observation distributions. And so at each step, we observe – at each step in the POMDP, if we're in some state ST, we observe some observation YT drawn from the observation distribution O subscript ST, that there's an index by what the current state is. And it turns out that computing the

---

optimal policy in a POMDP is an NP-hard problem. For the specific case of linear dynamical systems with the Kalman filter model, we have this strategy of computing the optimal policy assuming full observability and then estimating the states from the observations, and then plugging the two together.

That turns out to be optimal essentially for only that special case of a POMDP. In the more general case, that strategy of designing a controller assuming full observability and then just estimating the state and plugging the two together, for general POMDPs that same strategy is often a very reasonable strategy but is not always guaranteed to be optimal. Solving these problems in general, NP-hard. So what I want to do today is actually talk about a different class of reinforcement learning algorithms. These are called policy search algorithms. In particular, policy search algorithms can be applied equally well to MDPs, to fully observed Markov decision processes, or to these POMDPs, or to these partially observable MPDs. What I want to do now, I'll actually just describe policy search algorithms applied to MDPs, applied to the fully observable case. And in the end, I just briefly describe how you can take policy search algorithms and apply them to POMDPs. In the latter case, when you apply a policy search algorithm to a POMDP, it's going to be hard to guarantee that you get the globally optimal policy because solving POMDPs in general is NP-hard, but nonetheless policy search algorithms – it turns out to be I think one of the most effective classes of reinforcement learning algorithms, as well both for MDPs and for POMDPs.

So here's what we're going to do. In policy search, we're going to define of some set which I denote capital pi of policies, and our strategy is to search for a good policy lower pi into set capital pi. Just by analogy, I want to say – in the same way, back when we were talking about supervised learning, the way we defined the set capital pi of policies in the search for policy in this set capital pi is analogous to supervised learning where we defined a set script H of hypotheses and search – and would search for a good hypothesis in this policy script H. Policy search is sometimes also called direct policy search. To contrast this with the source of algorithms we've been talking about so far, in all the algorithms we've been talking about so far, we would try to find V star. We would try to find the optimal value function. And then we'd use V star – we'd use the optimal value function to then try to compute or try to approximate pi star. So all the approaches we talked about previously are strategy for finding a good policy. Once we compute the value function, then we go from that to policy. In contrast, in policy search algorithms and something that's called direct policy search algorithms, the idea is that we're going to quote "directly" try to approximate a good policy without going through the intermediate stage of trying to find the value function. Let's see. And also as I develop policy search – just one step that's sometimes slightly confusing. Making an analogy to supervised learning again, when we talked about logistic regression, I said we have input features X and some labels Y, and I sort of said let's approximate Y using the logistic function of the inputs X. And at least initially, the logistic function was sort of pulled out of the air.

In the same way, as I define policy search algorithms, there'll sort of be a step where I say, "Well, let's try to compute the actions. Let's try to approximate what a good action is using a logistic function of the state." So again, I'll sort of pull a function out of the air. I'll say, "Let's just choose a function, and that'll be our choice of the policy cost," and I'll say, "Let's take this input the state, and then we'll map it through logistic function, and then hopefully, we'll approximate what is a good function – excuse me, we'll approximate what is a good action using a logistic function of the state." So there's that sort of – the function of the choice of policy cost that's again a little bit arbitrary, but it's arbitrary as it was when we were talking about supervised learning. So to develop our first policy search algorithm, I'm actually gonna need the new definition. So our first policy search algorithm, we'll actually need to work with stochastic policies. What I mean by stochastic policy is there's going to be a function that maps from the space of states across actions. They're real numbers where pi of S comma A will be interpreted as the probability of taking this action A in sum state S. And so we have to add sum over A – In other words, for every state a stochastic policy specifies a probability distribution over the actions. So concretely, suppose you are executing some policy pi. Say I have some stochastic policy pi. I wanna execute the policy pi. What that means is that – in this example let's say I have three actions.

What that means is that suppose I'm in some state S. I would then compute pi of S comma A1, pi of S comma A2, pi of S comma A3, if I have a three action MDP. These will be three numbers that sum up to one, and then my chance of taking action A1 will be equal to this. My chance of taking action A2 will be

equal to pi of S comma A2. My chance of taking action A3 will be equal to this number. So that's what it means to execute a stochastic policy. So as a concrete example, just let me make this – the concept of why you wanna use stochastic policy is maybe a little bit hard to understand. So let me just go ahead and give one specific example of what a stochastic policy may look like. For this example, I'm gonna use the inverted pendulum as my motivating example. It's that problem of balancing a pole. We have an inverted pendulum that swings freely, and you want to move the cart left and right to keep the pole vertical. Let's say my actions – for today's example, I'm gonna use that angle to denote the angle of the pole phi. I have two actions where A1 is to accelerate left and A2 is to accelerate right. Actually, let me just write that the other way around. A1 is to accelerate right. A2 is to accelerate left. So let's see. Choose a reward function that penalizes the pole falling over whatever. And now let's come up with a stochastic policy for this problem. To come up with a class of stochastic policies really means coming up with some class of functions to approximate what action you want to take as a function of the state.

So here's my somewhat arbitrary choice. I'm gonna say that the probability of action A1, so pi of S comma A1, I'm gonna write as – okay? And I just chose the logistic function because it's a convenient function we've used a lot. So I'm gonna say that my policy is parameterized by a set of parameters theta, and for any given set of parameters theta, that gives me a stochastic policy. And if I'm executing that policy with parameters theta, that means that the chance of my choosing to a set of [inaudible] is given by this number. Because my chances of executing actions A1 or A2 must sum to one, this gives me pi of S A2. So just [inaudible], this means that when I'm in sum state S, I'm going to compute this number, compute one over one plus E to the minus state of transpose S. And then with this probability, I will execute the accelerate right action, and with one minus this probability, I'll execute the accelerate left action. And again, just to give you a sense of why this might be a reasonable thing to do, let's say my state vector is – this is [inaudible] state, and I added an extra one as an interceptor, just to give my logistic function an extra feature. If I choose my parameters and my policy to be say this, then that means that at any state, the probability of my taking action A1 – the probability of my taking the accelerate right action is this one over one plus E to the minus state of transpose S, which taking the inner product of theta and S, this just gives you phi, equals one over one plus E to the minus phi.

And so if I choose my parameters theta as follows, what that means is that just depending on the angle phi of my inverted pendulum, the chance of my accelerating to the right is just this function of the angle of my inverted pendulum. And so this means for example that if my inverted pendulum is leaning far over to the right, then I'm very likely to accelerate to the right to try to catch it. I hope the physics of this inverted pendulum thing make sense. If my pole's leaning over to the right, then I wanna accelerate to the right to catch it. And conversely if phi is negative, it's leaning over to the left, and I'll accelerate to the left to try to catch it. So this is one example for one specific choice of parameters theta. Obviously, this isn't a great policy because it ignores the rest of the features. Maybe if the cart is further to the right, you want it to be less likely to accelerate to the right, and you can capture that by changing one of these coefficients to take into account the actual position of the cart. And then depending on the velocity of the cart and the angle of velocity, you might want to change theta to take into account these other effects as well. Maybe if the pole's leaning far to the right, but is actually on its way to swinging back, it's specified to the angle of velocity, then you might be less worried about having to accelerate hard to the right. And so these are the sorts of behavior you can get by varying the parameters theta.

And so our goal is to tune the parameters theta – our goal in policy search is to tune the parameters theta so that when we execute the policy pi subscript theta, the pole stays up as long as possible. In other words, our goal is to maximize as a function of theta – our goal is to maximize the expected value of the payoff for when we execute the policy pi theta. We want to choose parameters theta to maximize that. Are there questions about the problem set up, and policy search and policy classes or anything? Yeah.

Student: In a case where we have more than two actions, would we use a different theta for each of the distributions, or still have the same parameters?

Instructor (Andrew Ng): Oh, yeah. Right. So what if we have more than two actions. It turns out you can choose almost anything you want for the policy class, but you have say a fixed number of discrete actions, I would sometimes use like a softmax parameterization. Similar to softmax regression that we saw

earlier in the class, you may say that − [inaudible] out of space. You may have a set of parameters theta 1 through theta D if you have D actions and − pi equals E to the theta I transpose S over − so that would be an example of a softmax parameterization for multiple actions. It turns out that if you have continuous actions, you can actually make this be a density over the actions A and parameterized by other things as well.

But the choice of policy class is somewhat up to you, in the same way that the choice of whether we chose to use a linear function or linear function with quadratic features or whatever in supervised learning that was sort of up to us. Anything else? Yeah.

Student:[Inaudible] stochastic?

Instructor (Andrew Ng): Yes.

Student: So is it possible to [inaudible] a stochastic policy using numbers [inaudible]?

Instructor (Andrew Ng):I see. Given that MDP has stochastic transition probabilities, is it possible to use [inaudible] policies and [inaudible] the stochasticity of the state transition probabilities. The answer is yes, but for the purposes of what I want to show later, that won't be useful. But formally, it is possible. If you already have a fixed − if you have a fixed policy, then you'd be able to do that. Anything else? Yeah. No, I guess even a [inaudible] class of policy can do that, but for the derivation later, I actually need to keep it separate. Actually, could you just − I know the concept of policy search is sometimes a little confusing. Could you just raise your hand if this makes sense? Okay. Thanks. So let's talk about an algorithm. What I'm gonna talk about − the first algorithm I'm going to present is sometimes called the reinforce algorithm. What I'm going to present it turns out isn't exactly the reinforce algorithm as it was originally presented by the author Ron Williams, but it sort of captures its essence.

Here's the idea. In the sequel − in what I'm about to do, I'm going to assume that S0 is some fixed initial state. Or it turns out if S0 is drawn from some fixed initial state distribution then everything else [inaudible], but let's just say S0 is some fixed initial state. So my goal is to maximize this expected sum [inaudible]. Given the policy and whatever else, drop that. So the random variables in this expectation is a sequence of states and actions: S0, A0, S1, A1, and so on, up to ST, AT are the random variables. So let me write out this expectation explicitly as a sum over all possible state and action sequences of that − so that's what an expectation is. It's the probability of the random variables times that. Let me just expand out this probability. So the probability of seeing this exact sequence of states and actions is the probability of the MDP starting in that state. If this is a deterministic initial state, then all the probability mass would be on one state. Otherwise, there's some distribution over initial states. Then times the probability that you chose action A0 from that state as zero, and then times the probability that the MDP's transition probabilities happen to transition you to state S1 where you chose action A0 to state S0, times the probability that you chose that and so on. The last term here is that, and then times that.

So what I did was just take this probability of seeing this sequence of states and actions, and then just [inaudible] explicitly or expanded explicitly like this. It turns out later on I'm going to need to write this sum of rewards a lot, so I'm just gonna call this the payoff from now. So whenever later in this lecture I write the word payoff, I just mean this sum. So our goal is to maximize the expected payoff, so our goal is to maximize this sum. Let me actually just skip ahead. I'm going to write down what the final answer is, and then I'll come back and justify the algorithm. So here's the algorithm. This is how we're going to update the parameters of the algorithm. We're going to sample a state action sequence. The way you do this is you just take your current stochastic policy, and you execute it in the MDP. So just go ahead and start from some initial state, take a stochastic action according to your current stochastic policy, see where the state transition probably takes you, and so you just do that for T times steps, and that's how you sample the state sequence. Then you compute the payoff, and then you perform this update.

So let's go back and figure out what this algorithm is doing. Notice that this algorithm performs stochastic updates because on every step it updates data according to this thing on the right hand side. This thing on the right hand side depends very much on your payoff and on the state action sequence you saw. Your state action sequence is random, so what I want to do is figure out − so on every step, I'll sort of take a step that's chosen randomly because it depends on this random state action sequence. So what I want to do is figure out on average how does it change the parameters theta. In particular, I want to know what is

the expected value of the change to the parameters. So I want to know what is the expected value of this change to my parameters theta. Our goal is to maximize the sum [inaudible] – our goal is to maximize the value of the payoff. So long as the updates on expectation are on average taking us uphill on the expected payoff, then we're happy. It turns out that this algorithm is a form of stochastic gradient ascent in which – remember when I talked about stochastic gradient descent for least squares regression, I said that you have some parameters – maybe you're trying to minimize a quadratic function. Then you may have parameters that will wander around randomly until it gets close to the optimum of the [inaudible] quadratic surface. It turns out that the reinforce algorithm will be very much like that. It will be a stochastic gradient ascent algorithm in which on every step – the step we take is a little bit random. It's determined by the random state action sequence, but on expectation this turns out to be essentially gradient ascent algorithm. And so we'll do something like this. It'll wander around randomly, but on average take you towards the optimum.

So let me go ahead and prove that now. Let's see. What I'm going to do is I'm going to derive a gradient ascent update rule for maximizing the expected payoff. Then I'll hopefully show that by deriving a gradient ascent update rule, I'll end up with this thing on expectation. So before I do the derivation, let me just remind you of the chain rule – the product rule for differentiation in which if I have a product of functions, then the derivative of the product is given by taking of the derivatives of these things one at a time. So first I differentiate with respect to F prime, leaving the other two fixed. Then I differentiate with respect to G, leaving the other two fixed. Then I differentiate with respect to H, so I get H prime leaving the other two fixed. So that's the product rule for derivatives. If you refer back to this equation where earlier we wrote out that the expected payoff by this equation, this sum over all the states of the probability times the payoff. So what I'm going to do is take the derivative of this expression with respect to the parameters theta because I want to do gradient ascent on this function. So I'm going to take the derivative of this function with respect to theta, and then try to go uphill on this function.

So using the product rule, when I take the derivative of this function with respect to theta what I get is – we'll end up with the sum of terms right there. There are a lot of terms here that depend on theta, and so what I'll end up with is I'll end up having a sum – having one term that corresponds to the derivative of this keeping everything else fixed, to have one term from the derivative of this keeping everything else fixed, and I'll have one term from the derivative of that last thing keeping everything else fixed. So just apply the product rule to this.

Let's write that down. So I have that – the derivative with respect to theta of the expected value of the payoff is – it turns out I can actually do this entire derivation in exactly four steps, but each of the steps requires a huge amount of writing, so I'll just start writing and see how that goes, but this is a four step derivation. So there's the sum over the state action sequences as we saw before. Close the bracket, and then times the payoff. So that huge amount of writing, that was just taking my previous formula and differentiating these terms that depend on theta one at a time. This was the term with the derivative of the first pi of theta S0 A0. So there's the first derivative term. There's the second one. Then you have plus dot, dot, dot, like in terms of [inaudible]. That's my last term. So that was step one of four. And so by algebra – let me just write this down and convince us all that it's true. This is the second of four steps in which it just convinced itself that if I expand out – take the sum and multiply it by that big product in front, then I get back that sum of terms I get. It's essentially – for example, when I multiply out, this product on top of this ratio, of this first fraction, then pi subscript theta S0 A0, that would cancel out this pi subscript theta S0 A0 and replace it with the derivative with respect to theta of pi theta S0 A0. So [inaudible] algebra was the second.

But that term on top is just what I worked out previously – was the joint probability of the state action sequence, and now I have that times that times the payoff. And so by the definition of expectation, this is just equal to that thing times the payoff. So this thing inside the expectation, this is exactly the step that we were taking in the inner group of our reinforce algorithm, roughly the reinforce algorithm. This proves that the expected value of our change to theta is exactly in the direction of the gradient of our expected payoff. That's how I started this whole derivation. I said let's look at our expected payoff and take the derivative of that with respect to theta. What we've proved is that on expectation, the step direction I'll take reinforce is exactly the gradient of the thing I'm trying to optimize. This shows that this algorithm is a stochastic

gradient ascent algorithm.

I wrote a lot. Why don't you take a minute to look at the equations and [inaudible] check if everything makes sense. I'll erase a couple of boards and then check if you have questions after that. Questions? Could you raise your hand if this makes sense? Great. Some of the comments – we talked about those value function approximation approaches where you approximate V star, then you go from V star to pi star. Then here was also policy search approaches, where you try to approximate the policy directly. So let's talk briefly about when either one may be preferable.

It turns out that policy search algorithms are especially effective when you can choose a simple policy class pi. So the question really is for your problem does there exist a simple function like a linear function or a logistic function that maps from features of the state to the action that works pretty well. So the problem with the inverted pendulum – this is quite likely to be true. Going through all the different choices of parameters, you can say things like if the pole's leaning towards the right, then accelerate towards the right to try to catch it. Thanks to the inverted pendulum, this is probably true. For lots of what's called low level control tasks, things like driving a car, the low level reflexes of do you steer your car left to avoid another car, do you steer your car left to follow the car road, flying a helicopter, again very short time scale types of decisions – I like to think of these as decisions like trained operator for like a trained driver or a trained pilot. It would almost be a reflex, these sorts of very quick instinctive things where you map very directly from the inputs, data, and action. These are problems for which you can probably choose a reasonable policy class like a logistic function or something, and it will often work well. In contrast, if you have problems that require long multistep reasoning, so things like a game of chess where you have to reason carefully about if I do this, then they'll do that, then they'll do this, then they'll do that. Those I think of as less instinctual, very high level decision making. For problems like that, I would sometimes use a value function approximation approaches instead.

Let me say more about this later. The last thing I want to do is actually tell you about – I guess just as a side comment, it turns out also that if you have POMDP, if you have a partially observable MDP – I don't want to say too much about this – it turns out that if you only have an approximation – let's call it S hat of the true state, and so this could be S hat equals S of T given T from Kalman filter – then you can still use these sorts of policy search algorithms where you can say pi theta of S hat comma A – There are various other ways you can use policy search algorithms for POMDPs, but this is one of them where if you only have estimates of the state, you can then choose a policy class that only looks at your estimate of the state to choose the action. By using the same way of estimating the states in both training and testing, this will usually do some – so these sorts of policy search algorithms can be applied often reasonably effectively to POMDPs as well. There's one more algorithm I wanna talk about, but some final words on the reinforce algorithm. It turns out the reinforce algorithm often works well but is often extremely slow. So it [inaudible] works, but one thing to watch out for is that because you're taking these gradient ascent steps that are very noisy, you're sampling a state action sequence, and then you're sort of taking a gradient ascent step in essentially a sort of random direction that only on expectation is correct.

The gradient ascent direction for reinforce can sometimes be a bit noisy, and so it's not that uncommon to need like a million iterations of gradient ascent, or ten million, or 100 million iterations of gradient ascent for reinforce [inaudible], so that's just something to watch out for. One consequence of that is in the reinforce algorithm – I shouldn't really call it reinforce. In what's essentially the reinforce algorithm, there's this step where you need to sample a state action sequence. So in principle you could do this on your own robot. If there were a robot you were trying to control, you can actually physically initialize in some state, pick an action and so on, and go from there to sample a state action sequence. But if you need to do this ten million times, you probably don't want to [inaudible] your robot ten million times. I personally have seen many more applications of reinforce in simulation. You can easily run ten thousand simulations or ten million simulations of your robot in simulation maybe, but you might not want to do that – have your robot physically repeat some action ten million times. So I personally have seen many more applications of reinforce to learn using a simulator than to actually do this on a physical device.

The last thing I wanted to do is tell you about one other algorithm, one final policy search algorithm. [Inaudible] the laptop display please. It's a policy search algorithm called Pegasus that's actually what we

use on our autonomous helicopter flight things for many years. There are some other things we do now. So here's the idea. There's a reminder slide on RL formalism. There's nothing here that you don't know, but I just want to pictorially describe the RL formalism because I'll use that later. I'm gonna draw the reinforcement learning picture as follows. The initialized [inaudible] system, say a helicopter or whatever in sum state S0, you choose an action A0, and then you'll say helicopter dynamics takes you to some new state S1, you choose some other action A1, and so on. And then you have some reward function, which you reply to the sequence of states you summed out, and that's your total payoff.

So this is just a picture I wanna use to summarize the RL problem. Our goal is to maximize the expected payoff, which is this, the expected sum of the rewards. And our goal is to learn the policy, which I denote by a green box. So our policy – and I'll switch back to deterministic policies for now. So my deterministic policy will be some function mapping from the states to the actions.

As a concrete example, you imagine that in the policy search setting, you may have a linear class of policies. So you may imagine that the action A will be say a linear function of the states, and your goal is to learn the parameters of the linear function. So imagine trying to do linear progression on policies, except you're trying to optimize the reinforcement learning objective. So just [inaudible] imagine that the action A is state of transpose S, and you go and policy search this to come up with good parameters theta so as to maximize the expected payoff. That would be one setting in which this picture applies. There's the idea. Quite often we come up with a model or a simulator for the MDP, and as before a model or a simulator is just a box that takes this input some state ST, takes this input some action AT, and then outputs some [inaudible] state ST plus one that you might want to take in the MDP. This ST plus one will be a random state. It will be drawn from the random state transition probabilities of MDP. This is important. Very important, ST plus one will be a random function ST and AT. In the simulator, this is [inaudible].

So for example, for autonomous helicopter flight, you [inaudible] build a simulator using supervised learning, an algorithm like linear regression [inaudible] to linear regression, so we can get a nonlinear model of the dynamics of what ST plus one is as a random function of ST and AT. Now once you have a simulator, given any fixed policy you can quite straightforwardly evaluate any policy in a simulator. Concretely, our goal is to find the policy pi mapping from states to actions, so the goal is to find the green box like that. It works well. So if you have any one fixed policy pi, you can evaluate the policy pi just using the simulator via the picture shown at the bottom of the slide. So concretely, you can take your initial state S0, feed it into the policy pi, your policy pi will output some action A0, you plug it in the simulator, the simulator outputs a random state S1, you feed S1 into the policy and so on, and you get a sequence of states S0 through ST that your helicopter flies through in simulation. Then sum up the rewards, and this gives you an estimate of the expected payoff of the policy.

This picture is just a fancy way of saying fly your helicopter in simulation and see how well it does, and measure the sum of rewards you get on average in the simulator. The picture I've drawn here assumes that you run your policy through the simulator just once. In general, you would run the policy through the simulator some number of times and then average to get an average over M simulations to get a better estimate of the policy's expected payoff. Now that I have a way – given any one fixed policy, this gives me a way to evaluate the expected payoff of that policy. So one reasonably obvious thing you might try to do is then just search for a policy, in other words search for parameters theta for your policy, that gives you high estimated payoff. Does that make sense? So my policy has some parameters theta, so my policy is my actions A are equal to theta transpose S say if there's a linear policy. For any fixed value of the parameters theta, I can evaluate – I can get an estimate for how good the policy is using the simulator. One thing I might try to do is search for parameters theta to try to maximize my estimated payoff. It turns out you can sort of do that, but that idea as I've just stated is hard to get to work. Here's the reason. The simulator allows us to evaluate policy, so let's search for policy of high value.

The difficulty is that the simulator is random, and so every time we evaluate a policy, we get back a very slightly different answer. So in the cartoon below, I want you to imagine that the horizontal axis is the space of policies. In other words, as I vary the parameters in my policy, I get different points on the horizontal axis here. As I vary the parameters theta, I get different policies, and so I'm moving along the X axis, and my total payoff I'm gonna plot on the vertical axis, and the red line in this cartoon is the expected payoff

of the different policies. My goal is to find the policy with the highest expected payoff. You could search for a policy with high expected payoff, but every time you evaluate a policy – say I evaluate some policy, then I might get a point that just by chance looked a little bit better than it should have. If I evaluate a second policy and just by chance it looked a little bit worse. I evaluate a third policy, fourth, sometimes you look here – sometimes I might actually evaluate exactly the same policy twice and get back slightly different answers just because my simulator is random, so when I apply the same policy twice in simulation, I might get back slightly different answers.

So as I evaluate more and more policies, these are the pictures I get. My goal is to try to optimize the red line. I hope you appreciate this is a hard problem, especially when all [inaudible] optimization algorithm gets to see are these black dots, and they don't have direct access to the red line. So when my input space is some fairly high dimensional space, if I have ten parameters, the horizontal axis would actually be a 10-D space, all I get are these noisy estimates of what the red line is. This is a very hard stochastic optimization problem. So it turns out there's one way to make this optimization problem much easier. Here's the idea. And the method is called Pegasus, which is an acronym for something. I'll tell you later. So the simulator repeatedly makes calls to a random number generator to generate random numbers RT, which are used to simulate the stochastic dynamics. What I mean by that is that the simulator takes this input of state and action, and it outputs the mixed state randomly, and if you peer into the simulator, if you open the box of the simulator and ask how is my simulator generating these random mixed states ST plus one, pretty much the only way to do so – pretty much the only way to write a simulator with random outputs is we're gonna make calls to a random number generator, and get random numbers, these RTs, and then you have some function that takes this input S0, A0, and the results of your random number generator, and it computes some mixed state as a function of the inputs and of the random number it got from the random number generator.

This is pretty much the only way anyone implements any random code, any code that generates random outputs. You make a call to a random number generator, and you compute some function of the random number and of your other inputs. The reason that when you evaluate different policies you get different answers is because every time you rerun the simulator, you get a different sequence of random numbers from the random number generator, and so you get a different answer every time, even if you evaluate the same policy twice. So here's the idea. Let's say we fix in advance the sequence of random numbers, choose R1, R2, up to RT minus one. Fix the sequence of random numbers in advance, and we'll always use the same sequence of random numbers to evaluate different policies. Go into your code and fix R1, R2, through RT minus one. Choose them randomly once and then fix them forever.

If you always use the same sequence of random numbers, then the system is no longer random, and if you evaluate the same policy twice, you get back exactly the same answer. And so these sequences of random numbers, [inaudible] call them scenarios, and Pegasus is an acronym for policy evaluation of gradient and search using scenarios. So when you do that, this is the picture you get. As before, the red line is your expected payoff, and by fixing the random numbers, you've defined some estimate of the expected payoff. And as you evaluate different policies, they're still only approximations to their true expected payoff, but at least now you have a deterministic function to optimize, and you can now apply your favorite algorithms, be it gradient ascent or some sort of local [inaudible] search to try to optimize the black curve. This gives you a much easier optimization problem, and you can optimize this to find hopefully a good policy. So this is the Pegasus policy search method.

So when I started to talk about reinforcement learning, I showed that video of a helicopter flying upside down. That was actually done using exactly method, using exactly this policy search algorithm. This seems to scale well even to fairly large problems, even to fairly high dimensional state spaces. Typically Pegasus policy search algorithms have been using – the optimization problem is still – is much easier than the stochastic version, but sometimes it's not entirely trivial, and so you have to apply this sort of method with maybe on the order of ten parameters or tens of parameters, so 30, 40 parameters, but not thousands of parameters, at least in these sorts of things with them.

Student: So is that method different than just assuming that you know your simulator exactly, just throwing away all the random numbers entirely?

Instructor (Andrew Ng):So is this different from assuming that we have a deterministic simulator? The answer is no. In the way you do this, for the sake of simplicity I talked about one sequence of random numbers. What you do is – so imagine that the random numbers are simulating different wind gusts against your helicopter. So what you want to do isn't really evaluate just against one pattern of wind gusts. What you want to do is sample some set of different patterns of wind gusts, and evaluate against all of them in average. So what you do is you actually sample say 100 – some number I made up like 100 sequences of random numbers, and every time you want to evaluate a policy, you evaluate it against all 100 sequences of random numbers and then average. This is in exactly the same way that on this earlier picture you wouldn't necessarily evaluate the policy just once. You evaluate it maybe 100 times in simulation, and then average to get a better estimate of the expected reward. In the same way, you do that here but with 100 fixed sequences of random numbers. Does that make sense? Any other questions?

Student:If we use 100 scenarios and get an estimate for the policy, [inaudible] 100 times [inaudible] random numbers [inaudible] won't you get similar ideas [inaudible]?

Instructor (Andrew Ng):Yeah. I guess you're right. So the quality – for a fixed policy, the quality of the approximation is equally good for both cases. The advantage of fixing the random numbers is that you end up with an optimization problem that's much easier. I have some search problem, and on the horizontal axis there's a space of control policies, and my goal is to find a control policy that maximizes the payoff.

The problem with this earlier setting was that when I evaluate policies I get these noisy estimates, and then it's just very hard to optimize the red curve if I have these points that are all over the place. And if I evaluate the same policy twice, I don't even get back the same answer. By fixing the random numbers, the algorithm still doesn't get to see the red curve, but at least it's now optimizing a deterministic function. That makes the optimization problem much easier. Does that make sense?

Student:So every time you fix the random numbers, you get a nice curve to optimize. And then you change the random numbers to get a bunch of different curves that are easy to optimize. And then you smush them together?

Instructor (Andrew Ng):Let's see. I have just one nice black curve that I'm trying to optimize.

Student:For each scenario.

Instructor (Andrew Ng):I see. So I'm gonna average over M scenarios, so I'm gonna average over 100 scenarios. So the black curve here is defined by averaging over a large set of scenarios. Does that make sense? So instead of only one – if the averaging thing doesn't make sense, imagine that there's just one sequence of random numbers. That might be easier to think about. Fix one sequence of random numbers, and every time I evaluate another policy, I evaluate against the same sequence of random numbers, and that gives me a nice deterministic function to optimize. Any other questions? The advantage is really that – one way to think about it is when I evaluate the same policy twice, at least I get back the same answer. This gives me a deterministic function mapping from parameters in my policy to my estimate of the expected payoff. That's just a function that I can try to optimize using the search algorithm. So we use this algorithm for inverted hovering, and again policy search algorithms tend to work well when you can find a reasonably simple policy mapping from the states to the actions. This is sort of especially the low level control tasks, which I think of as sort of reflexes almost.

Completely, if you want to solve a problem like Tetris where you might plan ahead a few steps about what's a nice configuration of the board, or something like a game of chess, or problems of long path plannings of go here, then go there, then go there, then sometimes you might apply a value function method instead. But for tasks like helicopter flight, for low level control tasks, for the reflexes of driving or controlling various robots, policy search algorithms were easier – we can sometimes more easily approximate the policy directly works very well. Some [inaudible] the state of RL today. RL algorithms are applied to a wide range of problems, and the key is really sequential decision making. The place where you think about applying reinforcement learning algorithm is when you need to make a decision, then another decision, then another decision, and some of your actions may have long-term consequences. I think that is the heart of RL's sequential decision making, where you make multiple decisions, and some of your actions may have long-term consequences. I've shown you a bunch of robotics examples. RL is also applied to thinks like medical decision making, where you may have a patient and you want to choose a sequence of treatments, and you do this now for

the patient, and the patient may be in some other state, and you choose to do that later, and so on.

It turns out there's a large community of people applying these sorts of tools to queues. So imagine you have a bank where you have people lining up, and after they go to one cashier, some of them have to go to the manager to deal with something else. You have a system of multiple people standing in line in multiple queues, and so how do you route people optimally to minimize the waiting time. And not just people, but objects in an assembly line and so on. It turns out there's a surprisingly large community working on optimizing queues. I mentioned game playing a little bit already. Things like financial decision making, if you have a large amount of stock, how do you sell off a large amount — how do you time the selling off of your stock so as to not affect market prices adversely too much? There are many operations research problems, things like factory automation. Can you design a factory to optimize throughput, or minimize cost, or whatever. These are all sorts of problems that people are applying reinforcement learning algorithms to.

Let me just close with a few robotics examples because they're always fun, and we just have these videos. This video was a work of Ziko Coulter and Peter Abiel, which is a PhD student here. They were working getting a robot dog to climb over difficult rugged terrain. Using a reinforcement learning algorithm, they applied an approach that's similar to a value function approximation approach, not quite but similar. They allowed the robot dog to sort of plan ahead multiple steps, and carefully choose his footsteps and traverse rugged terrain. This is really state of the art in terms of what can you get a robotic dog to do. Here's another fun one. It turns out that wheeled robots are very fuel-efficient. Cars and trucks are the most fuel-efficient robots in the world almost. Cars and trucks are very fuel-efficient, but the bigger robots have the ability to traverse more rugged terrain. So this is a robot — this is actually a small scale mockup of a larger vehicle built by Lockheed Martin, but can you come up with a vehicle that has wheels and has the fuel efficiency of wheeled robots, but also has legs so it can traverse obstacles. Again, using a reinforcement algorithm to design a controller for this robot to make it traverse obstacles, and somewhat complex gaits that would be very hard to design by hand, but by choosing a reward function, tell the robot this is a plus one reward that's top of the goal, and a few other things, it learns these sorts of policies automatically.

Last couple fun ones — I'll show you a couple last helicopter videos. This is the work of again PhD students here, Peter Abiel and Adam Coates who have been working on autonomous flight. I'll just let you watch. I'll just show you one more.

Student:[Inaudible] do this with a real helicopter [inaudible]?

Instructor (Andrew Ng): Not a full-size helicopter. Only small radio control helicopters.

Student:[Inaudible].

Instructor (Andrew Ng): Full-size helicopters are the wrong design for this. You shouldn't do this on a full-size helicopter. Many full-size helicopters would fall apart if you tried to do this. Let's see. There's one more.

Student: Are there any human [inaudible]?

Instructor (Andrew Ng): Yes, there are very good human pilots that can. This is just one more maneuver. That was kind of fun. So this is the work of Peter Abiel and Adam Coates. So that was it. That was all the technical material I wanted to present in this class. I guess you're all experts on machine learning now. Congratulations. And as I hope you've got the sense through this class that this is one of the technologies that's really having a huge impact on science in engineering and industry. As I said in the first lecture, I think many people use machine learning algorithms dozens of times a day without even knowing about it.

Based on the projects you've done, I hope that most of you will be able to imagine yourself going out after this class and applying these things to solve a variety of problems. Hopefully, some of you will also imagine yourselves writing research papers after this class, be it on a novel way to do machine learning, or on some way of applying machine learning to a problem that you care about. In fact, looking at project milestones, I'm actually sure that a large fraction of the projects in this class will be publishable, so I think that's great. I guess many of you will also go industry, make new products, and make lots of money using learning algorithms. Remember me if that happens. One of the things I'm excited about is through research or industry, I'm actually completely sure that the people in this class in the next few months will

apply machine learning algorithms to lots of problems in industrial management, and computer science, things like optimizing computer architectures, network security, robotics, computer vision, to problems in computational biology, to problems in aerospace, or understanding natural language, and many more things like that.

So right now I have no idea what all of you are going to do with the learning algorithms you learned about, but every time as I wrap up this class, I always feel very excited, and optimistic, and hopeful about the sorts of amazing things you'll be able to do. One final thing, I'll just give out this handout. One final thing is that machine learning has grown out of a larger literature on the AI where this desire to build systems that exhibit intelligent behavior and machine learning is one of the tools of AI, maybe one that's had a disproportionately large impact, but there are many other ideas in AI that I hope you go and continue to learn about. Fortunately, Stanford has one of the best and broadest sets of AI classes, and I hope that you take advantage of some of these classes, and go and learn more about AI, and more about other fields which often apply learning algorithms to problems in vision, problems in natural language processing in robotics, and so on.

So the handout I just gave out has a list of AI related courses. Just running down very quickly, I guess, CS221 is an overview that I teach. There are a lot of robotics classes also: 223A, 225A, 225B – many robotics class. There are so many applications of learning algorithms to robotics today. 222 and 227 are knowledge representation and reasoning classes. 228 – of all the classes on this list, 228, which Daphne Koller teaches, is probably closest in spirit to 229. This is one of the classes I highly recommend to all of my PhD students as well.

Many other problems also touch on machine learning. On the next page, courses on computer vision, speech recognition, natural language processing, various tools that aren't just machine learning, but often involve machine learning in many ways. Other aspects of AI, multi-agent systems taught by [inaudible]. EE364A is convex optimization. It's a class taught by Steve Boyd, and convex optimization came up many times in this class. If you want to become really good at it, EE364 is a great class. If you're interested in project courses, I also teach a project class next quarter where we spend the whole quarter working on research projects.

So I hope you go and take some more of those classes. In closing, let me just say this class has been really fun to teach, and it's very satisfying to me personally when we set these insanely difficult hallmarks, and then we'd see a solution, and I'd be like, "Oh my god. They actually figured that one out." It's actually very satisfying when I see that. Or looking at the milestones, I often go, "Wow, that's really cool. I bet this would be publishable." So I hope you take what you've learned, go forth, and do amazing things with learning algorithms. I know this is a heavy workload class, so thank you all very much for the hard work you've put into this class, and the hard work you've put into learning this material, and thank you very much for having been students in this class.

[End of Audio]
Duration: 78 minutes

Chapter 2

Lecture Notes

2.1 Machine Learning Lecture 1 Course Notes[1]

2.1.1 Supervised learning

Let's start by talking about a few examples of supervised learning problems. Suppose we have a dataset giving the living areas and prices of 47 houses from Portland, Oregon:
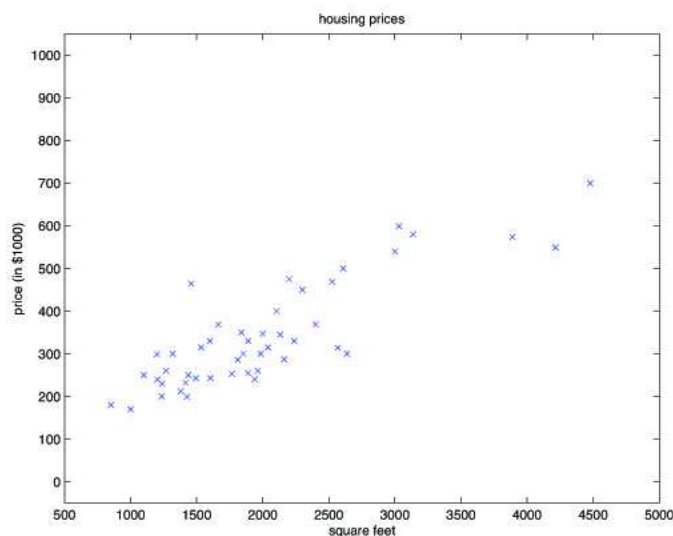
Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
$\vdots$	$\vdots$

Table 2.1

We can plot this data:

---

[1] This content is available online at <http://cnx.org/content/m45865/1.4/>.

Figure 2.1

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 $\left(x^{(i)}, y^{(i)}\right)$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $\{\left(x^{(i)}, y^{(i)}\right); i = 1, ..., m\}$—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 $h\left(x\right)$ 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:
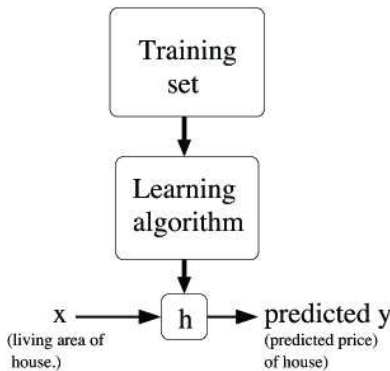
Figure 2.2

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.

2.1.2 Linear Regression

To make our housing example more interesting, let's consider a slightly richer dataset in which we also know the number of bedrooms in each house:

Living area (feet^2)	#bedrooms	Price (1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
$\vdots$	$\vdots$	$\vdots$

Table 2.2

Here, the x's are two-dimensional vectors in $\mathbb{R}^2$. For instance, $x_1^{(i)}$ is the living area of the i-th house in the training set, and $x_2^{(i)}$ is its number of bedrooms. (In general, when designing a learning problem, it will be up to you to decide what features to choose, so if you are out in Portland gathering housing data, you might also decide to include other features such as whether each house has a fireplace, the number of bathrooms, and so on. We'll say more about feature selection later, but for now let's take the features as given.)

To perform supervised learning, we must decide how we're going to represent functions /hypotheses h in a computer. As an initial choice, let's say we decide to approximate y as a linear function of x:

$$h_\theta\left(x\right) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \tag{2.1}$$

Here, the θ_i's are the **parameters** (also called **weights**) parameterizing the space of linear functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. When there is no risk of confusion, we will drop the θ subscript in $h_\theta(x)$, and write it more simply as $h(x)$. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the **intercept term**), so that

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x, \tag{2.2}$$

where on the right-hand side above we are viewing θ and x both as vectors, and here n is the number of input variables (not counting x_0).

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 $h\left(x^{(i)}\right)$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right)^2. \tag{2.3}$$

If you've seen linear regression before, you may recognize this as the familiar least-squares cost function that gives rise to the **ordinary least squares** regression model. Whether or not you have seen it previously, let's keep going, and we'll eventually show this to be a special case of a much broader family of algorithms.

2.1.3 LMS algorithm

We want to choose θ so as to minimize $J(\theta)$. To do so, let's use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, let's consider the **gradient descent** algorithm, which starts with some initial θ, and repeatedly performs the update:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta). \tag{2.4}$$

(This update is simultaneously performed for all values of $j = 0, ..., n$.) Here, α is called the **learning rate**. This is a very natural algorithm that repeatedly takes a step in the direction of steepest decrease of J.

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Let's first work it out for the case of if we have only one training example (x, y), so that we can neglect the sum in the definition of J. We have:

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} \left(h_\theta(x) - y \right)^2 \\
&= 2 \cdot \frac{1}{2} \left(h_\theta(x) - y \right) \cdot \frac{\partial}{\partial \theta_j} \left(h_\theta(x) - y \right) \\
&= \left(h_\theta(x) - y \right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{n} \theta_i x_i - y \right) \\
&= \left(h_\theta(x) - y \right) x_j
\end{aligned}
\tag{2.5}
$$

For a single training example, this gives the update rule:[2]

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta\left(x^{(i)}\right) \right) x_j^{(i)}. \tag{2.6}$$

The rule is called the **LMS** update rule (LMS stands for "least mean squares"), and is also known as the **Widrow-Hoff** learning rule. This rule has several properties that seem natural and intuitive. For instance,

---

[2]We use the notation "$a := b$" to denote an operation (in a computer program) in which we *set* the value of a variable a to be equal to the value of b. In other words, this operation overwrites a with the value of b. In contrast, we will write "$a = b$" when we are asserting a statement of fact, that the value of a is equal to the value of b.

the magnitude of the update is proportional to the **error** term $\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)$; 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 $h_\theta\left(x^{(i)}\right)$ has a large error (i.e., if it is very far from $y^{(i)}$).

We'd derived the LMS rule for when there was only a single training example. There are two ways to modify this method for a training set of more than one example. The first is replace it with the following algorithm:

1. Repeat until convergence {

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

2. }

The reader can easily verify that the quantity in the summation in the update rule above is just $\partial J\left(\theta\right)/\partial\theta_j$ (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method 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 (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.
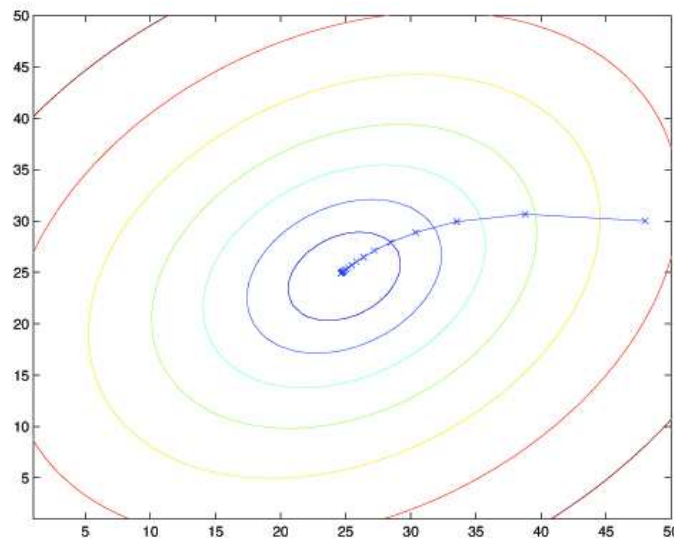

Figure 2.3

The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which 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 $h_\theta\left(x\right)$ as a function of x (area), along with the training data, we obtain the following figure:
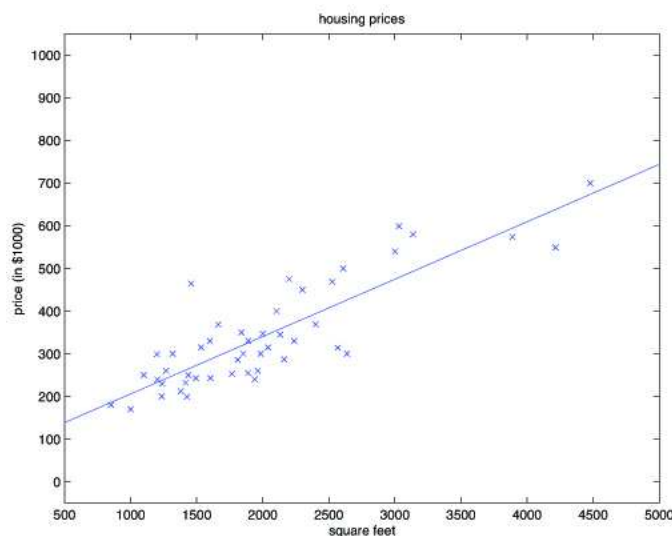
Figure 2.4

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:

1. Loop {
   a. for i=1 to m, {
      1. $\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta \left(x^{(i)} \right) \right) x_j^{(i)}$ (for every j).
   b. }
2. }

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.[3]) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

---

[3]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.

2.1.4 The normal equations

Gradient descent gives one way of minimizing J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In this method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. To enable us to do this without having to write reams of algebra and pages full of matrices of derivatives, let's introduce some notation for doing calculus with matrices.

2.1.4.1 Matrix derivatives

For a function $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ mapping from m-by-n matrices to the real numbers, we define the derivative of f with respect to A to be:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix} \tag{2.7}$$

Thus, the gradient $\nabla_A f(A)$ is itself an m-by-n matrix, whose (i, j)-element is $\partial f / \partial A_{ij}$. For example, suppose $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is a 2-by-2 matrix, and the function $f : \mathbb{R}^{2 \times 2} \mapsto \mathbb{R}$ is given by

$$f(A) = \frac{3}{2} A_{11} + 5 A_{12}^2 + A_{21} A_{22}. \tag{2.8}$$

Here, A_{ij} denotes the (i, j) entry of the matrix A. We then have

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10 A_{12} \\ A_{22} & A_{21} \end{bmatrix}. \tag{2.9}$$

We also introduce the **trace** operator, written "tr." For an n-by-n (square) matrix A, the trace of A is defined to be the sum of its diagonal entries:

$$\operatorname{tr} A = \sum_{i=1}^{n} A_{ii} \tag{2.10}$$

If a is a real number (i.e., a 1-by-1 matrix), then $\operatorname{tr} a = a$. (If you haven't seen this "operator notation" before, you should think of the trace of A as $\operatorname{tr}(A)$, or as application of the "trace" function to the matrix A. It's more commonly written without the parentheses, however.)

The trace operator has the property that for two matrices A and B such that AB is square, we have that $\operatorname{tr} AB = \operatorname{tr} BA$. (Check this yourself!) As corollaries of this, we also have, e.g.,

$$\begin{aligned} \operatorname{tr} ABC &= \operatorname{tr} CAB = \operatorname{tr} BCA, \\ \operatorname{tr} ABCD &= \operatorname{tr} DABC = \operatorname{tr} CDAB = \operatorname{tr} BCDA. \end{aligned} \tag{2.11}$$

The following properties of the trace operator are also easily verified. Here, A and B are square matrices, and a is a real number:

$$\begin{aligned} \operatorname{tr} A &= \operatorname{tr} A^T \\ \operatorname{tr}(A + B) &= \operatorname{tr} A + \operatorname{tr} B \\ \operatorname{tr} aA &= a \operatorname{tr} A \end{aligned} \tag{2.12}$$

We now state without proof some facts of matrix derivatives (we won't need some of these until later this quarter). (2.13) applies only to non-singular square matrices A, where $|A|$ denotes the determinant of A. We have:

$$
\begin{aligned}
\nabla_A \text{tr} AB &= & B^T \\
\nabla_{A^T} f(A) &= & \left(\nabla_A f(A)\right)^T \\
\nabla_A \text{tr} ABA^T C &= & CAB + C^T AB^T \\
\nabla_A |A| &= & |A|\left(A^{-1}\right)^T.
\end{aligned}
\tag{2.13}
$$

To make our matrix notation more concrete, let us now explain in detail the meaning of the first of these equations. Suppose we have some fixed matrix $B \in \mathbb{R}^{n \times m}$. We can then define a function $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ according to $f(A) = \text{tr} AB$. Note that this definition makes sense, because if $A \in \mathbb{R}^{m \times n}$, then AB is a square matrix, and we can apply the trace operator to it; thus, f does indeed map from $\mathbb{R}^{m \times n}$ to $\mathbb{R}$. We can then apply our definition of matrix derivatives to find $\nabla_A f(A)$, which will itself by an m-by-n matrix. (2.13) above states that the (i, j) entry of this matrix will be given by the (i, j)-entry of B^T, or equivalently, by B_{ji}.

The proofs of the first three equations in (2.13) are reasonably simple, and are left as an exercise to the reader. The fourth equation in (2.13) can be derived using the adjoint representation of the inverse of a matrix.[4]

2.1.4.2 Least squares revisited

Armed with the tools of matrix derivatives, let us now proceed to find in closed-form the value of θ that minimizes $J(\theta)$. We begin by re-writing J in matrix-vectorial notation.

Given a training set, define the **design matrix** X to be the m-by-n matrix (actually m-by-$n + 1$, if we include the intercept term) that contains the training examples' input values in its rows:

$$
X = \begin{bmatrix}
--- & (x(1))^T & --- \\
--- & (x(2))^T & --- \\
& \vdots & \\
--- & (x(m))^T & ---
\end{bmatrix}.
\tag{2.14}
$$

Also, let $\vec{y}$ be the m-dimensional vector containing all the target values from the training set:

$$
\vec{y} = \begin{bmatrix}
y^{(1)} \\
y^{(2)} \\
\vdots \\
y^{(m)}
\end{bmatrix}.
\tag{2.15}
$$

---

[4]If we define A' to be the matrix whose (i, j) element is $(-1)^{i+j}$ times the determinant of the square matrix resulting from deleting row i and column j from A, then it can be proved that $A^{-1} = \left(A'\right)^T / |A|$. (You can check that this is consistent with the standard way of finding A^{-1} when A is a 2-by-2 matrix. If you want to see a proof of this more general result, see an intermediate or advanced linear algebra text, such as Charles Curtis, 1991, *Linear Algebra*, Springer.) This shows that $A' = |A|\left(A^{-1}\right)^T$. Also, the determinant of a matrix can be written $|A| = \sum_j A_{ij} A'_{ij}$. Since $\left(A'\right)_{ij}$ does not depend on A_{ij} (as can be seen from its definition), this implies that $(\partial/\partial A_{ij}) |A| = A'_{ij}$. Putting all this together shows the result.

Now, since $h_\theta\left(x^{(i)}\right) = \left(x^{(i)}\right)^T \theta$, we can easily verify that

$$
\begin{aligned}
X\theta - \vec{y} &= \begin{bmatrix} \left(x^{(1)}\right)^T \theta \\ \vdots \\ \left(x^{(m)}\right)^T \theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \\
&= \begin{bmatrix} h_\theta\left(x^{(1)}\right) - y^{(1)} \\ \vdots \\ h_\theta\left(x^{(m)}\right) - y^{(m)} \end{bmatrix}.
\end{aligned}
\tag{2.16}
$$

Thus, using the fact that for a vector z, we have that $z^T z = \sum_i z_i^2$:

$$
\begin{aligned}
\frac{1}{2}\left(X\theta - \vec{y}\right)^T\left(X\theta - \vec{y}\right) &= \frac{1}{2}\sum_{i=1}^m \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 \\
&= J\left(\theta\right)
\end{aligned}
\tag{2.17}
$$

Finally, to minimize J, let's find its derivatives with respect to θ. Combining the second and third equation in (2.13), we find that

$$
\nabla_{A^T} \operatorname{tr} ABA^T C = B^T A^T C^T + BA^T C
\tag{2.18}
$$

Hence,

$$
\begin{aligned}
\nabla_\theta J\left(\theta\right) &= \nabla_\theta \frac{1}{2}\left(X\theta - \vec{y}\right)^T\left(X\theta - \vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T \vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \operatorname{tr}\left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T \vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\operatorname{tr}\theta^T X^T X\theta - 2\operatorname{tr}\vec{y}^T X\theta\right) \\
&= \frac{1}{2}\left(X^T X\theta + X^T X\theta - 2X^T \vec{y}\right) \\
&= X^T X\theta - X^T \vec{y}
\end{aligned}
\tag{2.19}
$$

In the third step, we used the fact that the trace of a real number is just the real number; the fourth step used the fact that $\operatorname{tr}A = \operatorname{tr}A^T$, and the fifth step used Equation (2.18) with $A^T = \theta$, $B = B^T = X^T X$, and $C = I$, and Equation (2.13). To minimize J, we set its derivatives to zero, and obtain the **normal equations**:

$$
X^T X\theta = X^T \vec{y}
\tag{2.20}
$$

Thus, the value of θ that minimizes $J\left(\theta\right)$ is given in closed form by the equation

$$
\theta = \left(X^T X\right)^{-1} X^T \vec{y}.
\tag{2.21}
$$

2.1.5 Probabilistic interpretation

When faced with a regression problem, why might linear regression, and specifically why might the least-squares cost function J, be a reasonable choice? In this section, we will give a set of probabilistic assumptions, under which least-squares regression is derived as a very natural algorithm.

Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}, \tag{2.22}$$

where $\varepsilon^{(i)}$ is an error term that captures either unmodeled effects (such as if there are some features very pertinent to predicting housing price, but that we'd left out of the regression), or random noise. Let us further assume that the $\varepsilon^{(i)}$ are distributed IID (independently and identically distributed) according to a Gaussian distribution (also called a Normal distribution) with mean zero and some variance σ^2. We can write this assumption as "$\varepsilon^{(i)} \sim \mathcal{N}\left(0, \sigma^2\right)$." I.e., the density of $\varepsilon^{(i)}$ is given by

$$p\left(\varepsilon^{(i)}\right) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(\varepsilon^{(i)}\right)^2}{2\sigma^2}\right). \tag{2.23}$$

This implies that

$$p\left(y^{(i)}|x^{(i)};\theta\right) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right). \tag{2.24}$$

The notation "$p\left(y^{(i)}|x^{(i)};\theta\right)$" indicates that this is the distribution of $y^{(i)}$ given $x^{(i)}$ and parameterized by θ. Note that we should not condition on θ ("$p\left(y^{(i)}|x^{(i)},\theta\right)$"), since θ is not a random variable. We can also write the distribution of $y^{(i)}$ as as $y^{(i)} \mid x^{(i)}; \theta \sim \mathcal{N}\left(\theta^T x^{(i)}, \sigma^2\right)$.

Given X (the design matrix, which contains all the $x^{(i)}$'s) and θ, what is the distribution of the $y^{(i)}$'s? The probability of the data is given by $p\left(\vec{y} \mid X;\theta\right)$. This quantity is typically viewed a function of $\vec{y}$ (and perhaps X), for a fixed value of θ. When we wish to explicitly view this as a function of θ, we will instead call it the **likelihood** function:

$$L\left(\theta\right) = L\left(\theta; X, \vec{y}\right) = p\left(\vec{y} \mid X;\theta\right). \tag{2.25}$$

Note that by the independence assumption on the $\varepsilon^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this can also be written

$$\begin{aligned} L\left(\theta\right) &= \prod_{i=1}^{m} p\left(y^{(i)} \mid x^{(i)};\theta\right) \\ &= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right). \end{aligned} \tag{2.26}$$

Now, given this probabilistic model relating the $y^{(i)}$'s and the $x^{(i)}$'s, what is a reasonable way of choosing our best guess of the parameters θ? The principal of **maximum likelihood** says that we should should choose θ so as to make the data as high probability as possible. I.e., we should choose θ to maximize $L\left(\theta\right)$.

Instead of maximizing $L\left(\theta\right)$, we can also maximize any strictly increasing function of $L\left(\theta\right)$. In particular, the derivations will be a bit simpler if we instead maximize the **log likelihood**$\ell\left(\theta\right)$:

$$\begin{aligned} \ell\left(\theta\right) &= logL\left(\theta\right) \\ &= log\prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^{m} log\frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &= m log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - \theta^T x^{(i)}\right)^2. \end{aligned} \tag{2.27}$$

Hence, maximizing $\ell\left(\theta\right)$ gives the same answer as minimizing

$$\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - \theta^T x^{(i)}\right)^2, \tag{2.28}$$

which we recognize to be $J(\theta)$, our original least-squares cost function.

To summarize: Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. This is thus one set of assumptions under which least-squares regression can be justified as a very natural method that's just doing maximum likelihood estimation. (Note however that the probabilistic assumptions are by no means *necessary* for least-squares to be a perfectly good and rational procedure, and there may—and indeed there are—other natural assumptions that can also be used to justify it.)

Note also that, in our previous discussion, our final choice of θ did not depend on what was σ^2, and indeed we'd have arrived at the same result even if σ^2 were unknown. We will use this fact again later, when we talk about the exponential family and generalized linear models.

2.1.6 Locally weighted linear regression

Consider the problem of predicting y from $x \in \mathbb{R}$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.
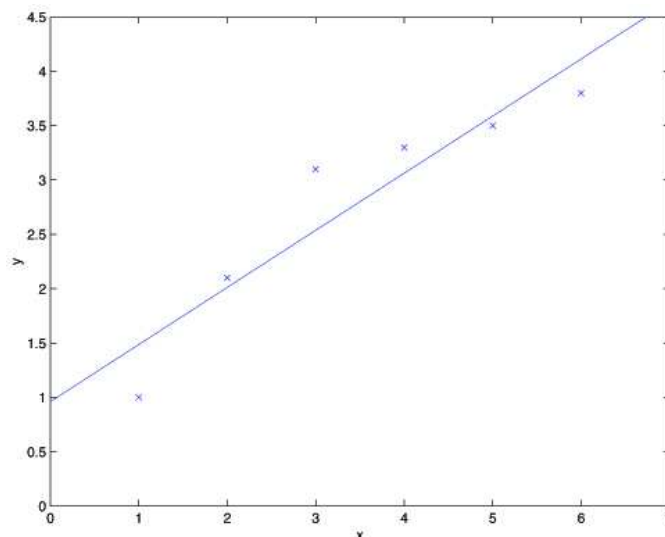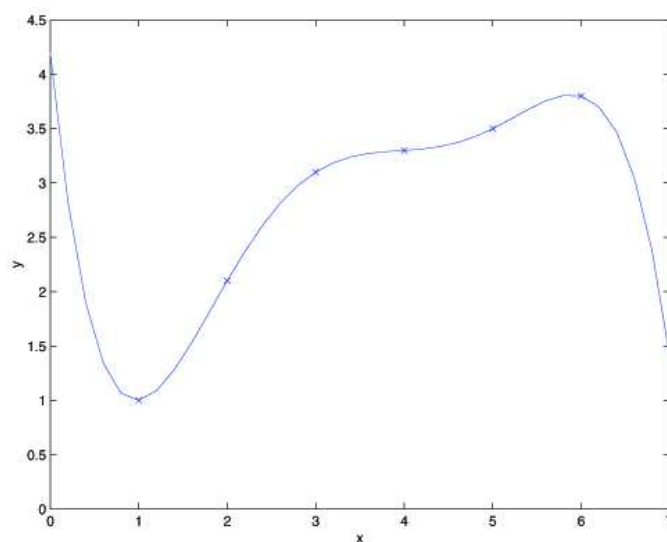

Figure 2.5
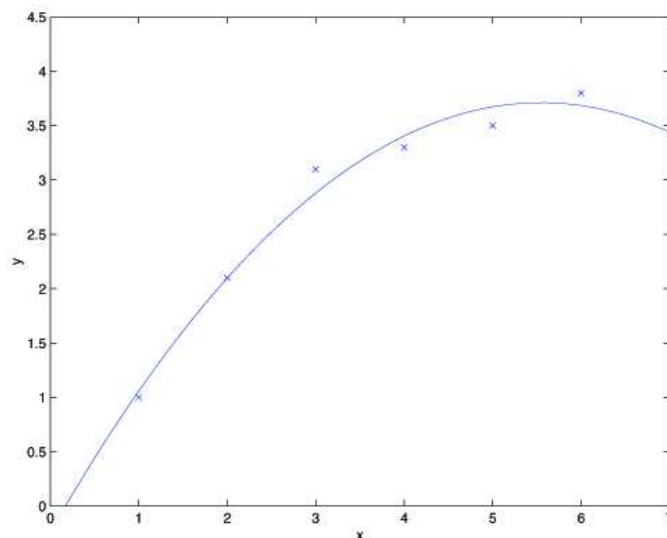
Figure 2.6

Figure 2.7

Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data. (See middle figure) Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5-th order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an

instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**. (Later in this class, when we talk about learning theory we'll formalize some of these notions, and also define more carefully just what it means for a hypothesis to be good or bad.)

As discussed previously, and as shown in the example above, the choice of features is important to ensuring good performance of a learning algorithm. (When we talk about model selection, we'll also see algorithms for automatically choosing a good set of features.) In this section, let us talk briefly talk about the locally weighted linear regression (LWR) algorithm which, assuming there is sufficient training data, makes the choice of features less critical. This treatment will be brief, since you'll get a chance to explore some of the properties of the LWR algorithm yourself in the homework.

In the original linear regression algorithm, to make a prediction at a query point x (i.e., to evaluate $h(x)$), we would:

1. Fit θ to minimize $\sum_i \left(y^{(i)} - \theta^T x^{(i)} \right)^2$.
2. Output $\theta^T x$.

In contrast, the locally weighted linear regression algorithm does the following:

1. Fit θ to minimize $\sum_i w^{(i)} \left(y^{(i)} - \theta^T x^{(i)} \right)^2$.
2. Output $\theta^T x$.

Here, the $w^{(i)}$'s are non-negative valued **weights**. Intuitively, if $w^{(i)}$ is large for a particular value of i, then in picking θ, we'll try hard to make $\left(y^{(i)} - \theta^T x^{(i)} \right)^2$ small. If $w^{(i)}$ is small, then the $\left(y^{(i)} - \theta^T x^{(i)} \right)^2$ error term will be pretty much ignored in the fit.

A fairly standard choice for the weights is[5]

$$w^{(i)} = exp\left(-\frac{\left(x^{(i)} - x \right)^2}{2\tau^2} \right) \tag{2.29}$$

Note that the weights depend on the particular point x at which we're trying to evaluate x. Moreover, if $|x^{(i)} - x|$ is small, then $w^{(i)}$ is close to 1; and if $|x^{(i)} - x|$ is large, then $w^{(i)}$ is small. Hence, θ is chosen giving a much higher "weight" to the (errors on) training examples close to the query point x. (Note also that while the formula for the weights takes a form that is cosmetically similar to the density of a Gaussian distribution, the $w^{(i)}$'s do not directly have anything to do with Gaussians, and in particular the $w^{(i)}$ are not random variables, normally distributed or otherwise.) The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x; τ is called the **bandwidth** parameter, and is also something that you'll get to experiment with in your homework.

Locally weighted linear regression is the first example we're seeing of a **non-parametric** algorithm. The (unweighted) linear regression algorithm that we saw earlier is known as a **parametric** learning algorithm, because it has a fixed, finite number of parameters (the θ_i's), which are fit to the data. Once we've fit the θ_i's and stored them away, we no longer need to keep the training data around to make future predictions. In contrast, to make predictions using locally weighted linear regression, we need to keep the entire training set around. The term "non-parametric" (roughly) refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis h grows linearly with the size of the training set.

2.1.7 Classification and logistic regression

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

---

[5]If x is vector-valued, this is generalized to be $w^{(i)} = exp\left(-\left(x^{(i)} - x \right)^T \left(x^{(i)} - x \right) / \left(2\tau^2 \right) \right)$, or $w^{(i)} = exp\left(-\left(x^{(i)} - x \right)^T \Sigma^{-1} \left(x^{(i)} - x \right) / 2 \right)$, for an appropriate choice of τ or Σ.

the **binary classification** 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.

2.1.8 Logistic regression

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. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0,1\}$.

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

$$h_\theta(x) = g\left(\theta^T x\right) = \frac{1}{1 + e^{-\theta^T x}}, \tag{2.30}$$

where

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.31}$$

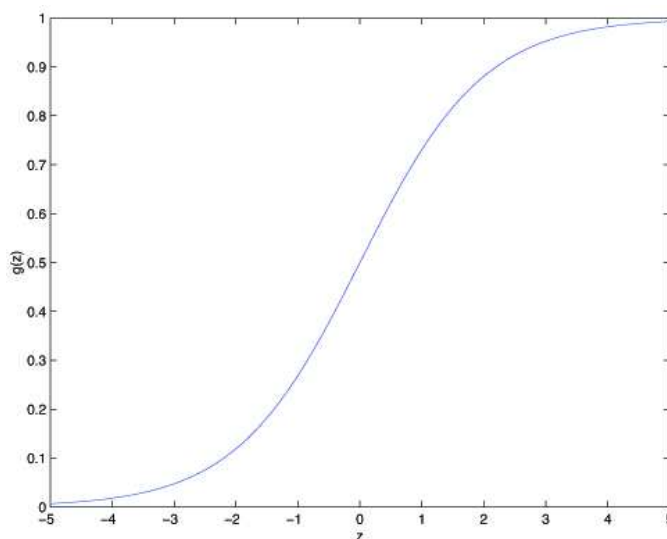is called the **logistic function** or the **sigmoid function**. Here is a plot showing $g(z)$:

Figure 2.8

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 $\theta^T x = \theta_0 + \sum_{j=1}^n \theta_j x_j$.

For now, let's take the choice of g as given. Other functions that smoothly increase from 0 to 1 can also be used, but for a couple of reasons that we'll see later (when we talk about GLMs, and when we talk about

generative learning algorithms), the choice of the logistic function is a fairly natural one. Before moving on, here's a useful property of the derivative of the sigmoid function, which we write as g':

$$
\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\
&= g(z)\left(1-g(z)\right).
\end{aligned}
\tag{2.32}
$$

So, given the logistic regression model, how do we fit θ for it? Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, let's endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood.

Let us assume that

$$
\begin{aligned}
P(y=1\mid x;\theta) &= h_\theta(x) \\
P(y=0\mid x;\theta) &= 1-h_\theta(x)
\end{aligned}
\tag{2.33}
$$

Note that this can be written more compactly as

$$
p(y\mid x;\theta) = (h_\theta(x))^y(1-h_\theta(x))^{1-y}
\tag{2.34}
$$

Assuming that the m training examples were generated independently, we can then write down the likelihood of the parameters as

$$
\begin{aligned}
L(\theta) &= p\left(\vec{y}\mid X;\theta\right) \\
&= \prod_{i=1}^{m}p\left(y^{(i)}\mid x^{(i)};\theta\right) \\
&= \prod_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)\right)^{y^{(i)}}\left(1-h_\theta\left(x^{(i)}\right)\right)^{1-y^{(i)}}
\end{aligned}
\tag{2.35}
$$

As before, it will be easier to maximize the log likelihood:

$$
\begin{aligned}
\ell(\theta) &= \log L(\theta) \\
&= \sum_{i=1}^{m}y^{(i)}\log h\left(x^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)}\right)\right)
\end{aligned}
\tag{2.36}
$$

How do we maximize the likelihood? Similar to our derivation in the case of linear regression, we can use gradient ascent. Written in vectorial notation, our updates will therefore 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.) Let's start by working with just one training example (x,y), and take derivatives to derive the stochastic gradient ascent rule:

$$
\begin{aligned}
\frac{\partial}{\partial\theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^Tx)}-(1-y)\frac{1}{1-g(\theta^Tx)}\right)\frac{\partial}{\partial\theta_j}g\left(\theta^Tx\right) \\
&= \left(y\frac{1}{g(\theta^Tx)}-(1-y)\frac{1}{1-g(\theta^Tx)}\right)g\left(\theta^Tx\right)\left(1-g\left(\theta^Tx\right)\frac{\partial}{\partial\theta_j}\theta^Tx\right. \\
&= \left(y\left(1-g\left(\theta^Tx\right)\right)-(1-y)g\left(\theta^Tx\right)\right)x_j \\
&= \left(y-h_\theta(x)\right)x_j
\end{aligned}
\tag{2.37}
$$

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

$$
\theta_j := \theta_j+\alpha\left(y^{(i)}-h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}
\tag{2.38}
$$

If we compare this to the LMS update rule, we see that it looks identical; but this is *not* the same algorithm, because $h_\theta\left(x^{(i)}\right)$ is now defined as a non-linear function of $\theta^T x^{(i)}$. Nonetheless, it's a little surprising that we end up with the same update rule for a rather different algorithm and learning problem. Is this coincidence, or is there a deeper reason behind this? We'll answer this when get get to GLM models. (See also the extra credit problem on Q3 of problem set 1.)

2.1.9 Digression: The perceptron learning algorithm

We now digress to talk briefly about an algorithm that's of some historical interest, and that we will also return to later when we talk about learning theory. Consider modifying the logistic regression method to "force" it to output values that are either 0 or 1 or exactly. To do so, it seems natural to change the definition of g to be the threshold function:

$$g\left(z\right) = \{ \begin{array}{ll} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{array} \tag{2.39}$$

If we then let $h_\theta\left(x\right) = g\left(\theta^T x\right)$ as before but using this modified definition of g, and if we use the update rule

$$\theta_j := \theta_j + \alpha\left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right)x_j^{(i)}. \tag{2.40}$$

then we have the **perceptron learning algorithm**.

In the 1960s, this "perceptron" was argued to be a rough model for how individual neurons in the brain work. Given how simple the algorithm is, it will also provide a starting point for our analysis when we talk about learning theory later in this class. Note however that even though the perceptron may be cosmetically similar to the other algorithms we talked about, it is actually a very different type of algorithm than logistic regression and least squares linear regression; in particular, it is difficult to endow the perceptron's predictions with meaningful probabilistic interpretations, or derive the perceptron as a maximum likelihood estimation algorithm.

2.1.10 Another algorithm for maximizing $\ell\left(\theta\right)$

Returning to logistic regression with $g\left(z\right)$ being the sigmoid function, let's now talk about a different algorithm for maximizing $\ell\left(\theta\right)$.

To get us started, let's consider Newton's method for finding a zero of a function. Specifically, suppose we have some function $f : \mathbb{R} \mapsto \mathbb{R}$, and we wish to find a value of θ so that $f\left(\theta\right) = 0$. Here, $\theta \in \mathbb{R}$ is a real number. Newton's method performs the following update:

$$\theta := \theta - \frac{f\left(\theta\right)}{f^{,}\left(\theta\right)}. \tag{2.41}$$

This method has a natural interpretation in which we can think of it as approximating the function f via a linear function that is tangent to f at the current guess θ, solving for where that linear function equals to zero, and letting the next guess for θ be where that linear function is zero.

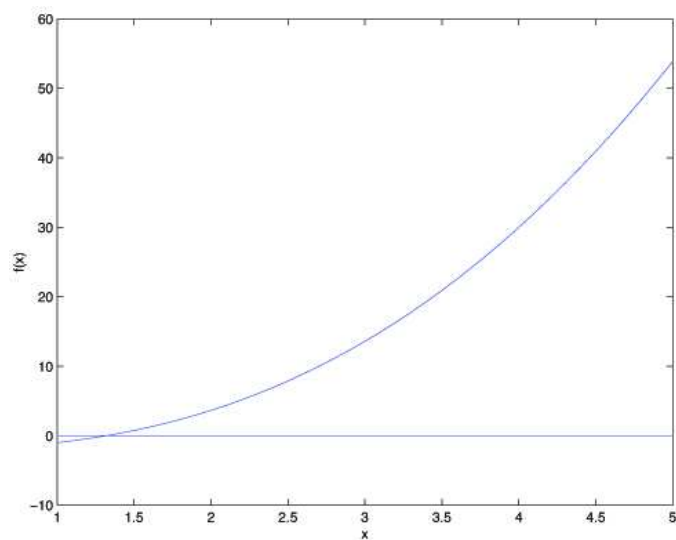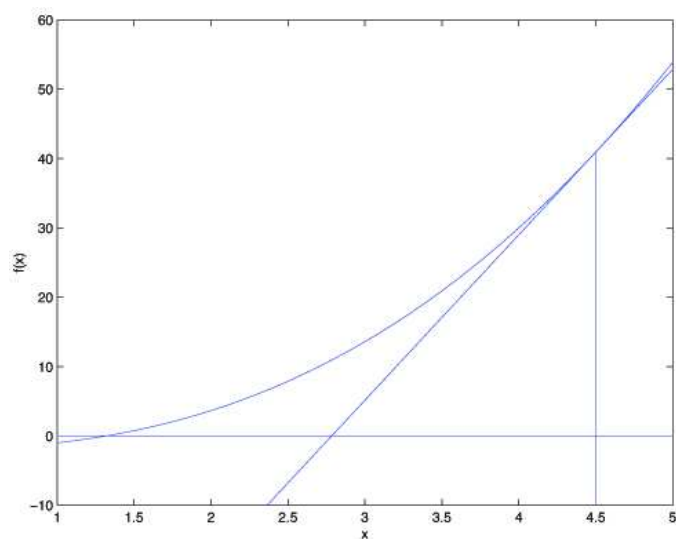Here's a picture of the Newton's method in action:
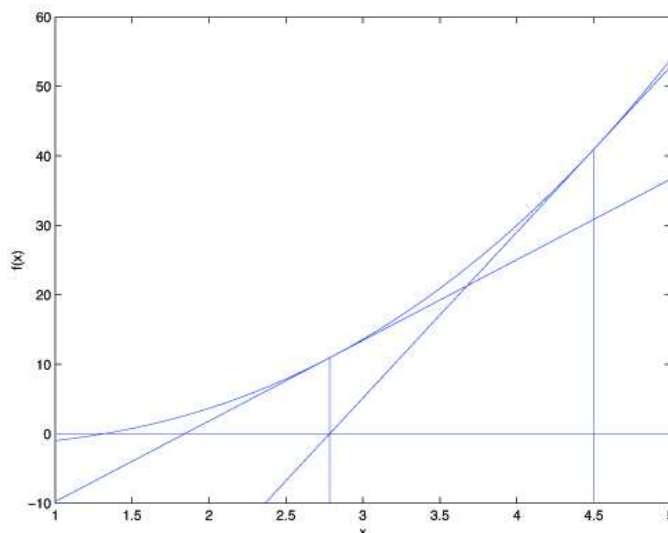
Figure 2.9

Figure 2.10

Figure 2.11

In the leftmost figure, we see the function f plotted along with the line $y = 0$. We're trying to find θ so that $f(\theta) = 0$; the value of θ that achieves this is about 1.3. Suppose we initialized the algorithm with $\theta = 4.5$. Newton's method then fits a straight line tangent to f at $\theta = 4.5$, and solves for the where that line evaluates to 0. (Middle figure.) This give us the next guess for θ, which is about 2.8. The rightmost figure shows the result of running one more iteration, which the updates θ to about 1.8. After a few more iterations, we rapidly approach $\theta = 1.3$.

Newton's method gives a way of getting to $f(\theta) = 0$. What if we want to use it to maximize some function ℓ? The maxima of ℓ correspond to points where its first derivative $\ell'(\theta)$ is zero. So, by letting $f(\theta) = \ell'(\theta)$, we can use the same algorithm to maximize ℓ, and we obtain update rule:

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}. \tag{2.42}$$

(Something to think about: How would this change if we wanted to use Newton's method to minimize rather than maximize a function?)

Lastly, in our logistic regression setting, θ is vector-valued, so we need to generalize Newton's method to this setting. The generalization of Newton's method to this multidimensional setting (also called the Newton-Raphson method) is given by

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta). \tag{2.43}$$

Here, $\nabla_\theta \ell(\theta)$ is, as usual, the vector of partial derivatives of $\ell(\theta)$ with respect to the θ_i's; and H is an n-by-n matrix (actually, $n + 1$-by-$n + 1$, assuming that we include the intercept term) called the **Hessian**, whose entries are given by

$$H_{ij} = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}. \tag{2.44}$$

Newton's method typically enjoys faster convergence than (batch) gradient descent, and requires many fewer iterations to get very close to the minimum. One iteration of Newton's can, however, be more expensive than one iteration of gradient descent, since it requires finding and inverting an n-by-n Hessian; but so long

as n is not too large, it is usually much faster overall. When Newton's method is applied to maximize the logistic regression log likelihood function $\ell\left(\theta\right)$, the resulting method is also called **Fisher scoring**.

2.1.11 Generalized Linear Models[6]

So far, we've seen a regression example, and a classification example. In the regression example, we had $y|x;\theta \sim \mathcal{N}\left(\mu, \sigma^2\right)$, and in the classification one, $y|x;\theta \sim \text{Bernoulli}\left(\Phi\right)$, for some appropriate definitions of μ and Φ as functions of x and θ. In this section, we will show that both of these methods are special cases of a broader family of models, called Generalized Linear Models (GLMs). We will also show how other models in the GLM family can be derived and applied to other classification and regression problems.

2.1.12 The exponential family

To work our way up to GLMs, we will begin by defining exponential family distributions. We say that a class of distributions is in the exponential family if it can be written in the form

$$p\left(y;\eta\right) = b\left(y\right)exp\left(\eta^T T\left(y\right) - a\left(\eta\right)\right) \tag{2.45}$$

Here, η is called the **natural parameter** (also called the **canonical parameter**) of the distribution; $T\left(y\right)$ is the **sufficient statistic** (for the distributions we consider, it will often be the case that $T\left(y\right) = y$); and $a\left(\eta\right)$ is the **log partition function**. The quantity $e^{-a(\eta)}$ essentially plays the role of a normalization constant, that makes sure the distribution $p\left(y;\eta\right)$ sums/integrates over y to 1.

A fixed choice of T, a and b defines a *family* (or set) of distributions that is parameterized by η; as we vary η, we then get different distributions within this family.

We now show that the Bernoulli and the Gaussian distributions are examples of exponential family distributions. The Bernoulli distribution with mean Φ, written $\text{Bernoulli}\left(\Phi\right)$, specifies a distribution over $y \in \{0, 1\}$, so that $p\left(y = 1; \Phi\right) = \Phi$; $p\left(y = 0; \Phi\right) = 1 - \Phi$. As we vary Φ, we obtain Bernoulli distributions with different means. We now show that this class of Bernoulli distributions, ones obtained by varying Φ, is in the exponential family; i.e., that there is a choice of T, a and b so that Equation (2.45) becomes exactly the class of Bernoulli distributions.

We write the Bernoulli distribution as:

$$
\begin{aligned}
p\left(y;\Phi\right) &= & \Phi^y(1-\Phi)^{1-y} \\
&= & exp\left(ylog\Phi + (1-y)\,log\left(1-\Phi\right)\right) \\
&= & exp\left(\left(log\left(\frac{\Phi}{1-\Phi}\right)\right)y + log\left(1-\Phi\right)\right).
\end{aligned}
\tag{2.46}
$$

Thus, the natural parameter is given by $\eta = log\left(\Phi/\left(1-\Phi\right)\right)$. Interestingly, if we invert this definition for η by solving for Φ in terms of η, we obtain $\Phi = 1/\left(1 + e^{-\eta}\right)$. This is the familiar sigmoid function! This will come up again when we derive logistic regression as a GLM. To complete the formulation of the Bernoulli distribution as an exponential family distribution, we also have

$$
\begin{aligned}
T\left(y\right) &= & y \\
a\left(\eta\right) &= & -log\left(1-\Phi\right) \\
&= & log\left(1 + e^\eta\right) \\
b\left(y\right) &= & 1
\end{aligned}
\tag{2.47}
$$

This shows that the Bernoulli distribution can be written in the form of Equation (2.45), using an appropriate choice of T, a and b.

---

[6]The presentation of the material in this section takes inspiration from Michael I. Jordan, *Learning in graphical models* (unpublished book draft), and also McCullagh and Nelder, *Generalized Linear Models (2nd ed.)* .

Let's now move on to consider the Gaussian distribution. Recall that, when deriving linear regression, the value of σ^2 had no effect on our final choice of θ and $h_\theta(x)$. Thus, we can choose an arbitrary value for σ^2 without changing anything. To simplify the derivation below, let's set $\sigma^2 = 1$.[7] We then have:

$$
\begin{aligned}
p(y;\mu) &= \frac{1}{\sqrt{2\pi}} exp\left(-\frac{1}{2}(y-\mu)^2\right) \\
&= \frac{1}{\sqrt{2\pi}} exp\left(-\frac{1}{2}y^2\right) \cdot exp\left(\mu y - \frac{1}{2}\mu^2\right)
\end{aligned}
\tag{2.48}
$$

Thus, we see that the Gaussian is in the exponential family, with

$$
\begin{aligned}
\eta &= \mu \\
T(y) &= y \\
a(\eta) &= \mu^2/2 \\
&= \eta^2/2 \\
b(y) &= \left(1/\sqrt{2\pi}\right) exp\left(-y^2/2\right).
\end{aligned}
\tag{2.49}
$$

There're many other distributions that are members of the exponential family: The multinomial (which we'll see later), the Poisson (for modelling count-data; also see the problem set); the gamma and the exponential (for modelling continuous, non-negative random variables, such as time-intervals); the beta and the Dirichlet (for distributions over probabilities); and many more. In the next section, we will describe a general "recipe" for constructing models in which y (given x and θ) comes from any of these distributions.

2.1.13 Constructing GLMs

Suppose you would like to build a model to estimate the number y of customers arriving in your store (or number of page-views on your website) in any given hour, based on certain features x such as store promotions, recent advertising, weather, day-of-week, etc. We know that the Poisson distribution usually gives a good model for numbers of visitors. Knowing this, how can we come up with a model for our problem? Fortunately, the Poisson is an exponential family distribution, so we can apply a Generalized Linear Model (GLM). In this section, we will we will describe a method for constructing GLM models for problems such as these.

More generally, consider a classification or regression problem where we would like to predict the value of some random variable y as a function of x. To derive a GLM for this problem, we will make the following three assumptions about the conditional distribution of y given x and about our model:

1. $y \mid x;\theta \sim \text{ExponentialFamily}(\eta)$. I.e., given x and θ, the distribution of y follows some exponential family distribution, with parameter η.

2. Given x, our goal is to predict the expected value of $T(y)$ given x. In most of our examples, we will have $T(y) = y$, so this means we would like the prediction $h(x)$ output by our learned hypothesis h to satisfy $h(x) = E[y|x]$. (Note that this assumption is satisfied in the choices for $h_\theta(x)$ for both logistic regression and linear regression. For instance, in logistic regression, we had $h_\theta(x) = p(y=1|x;\theta) = 0 \cdot p(y=0|x;\theta) + 1 \cdot p(y=1|x;\theta) = E[y|x;\theta]$.)

3. The natural parameter η and the inputs x are related linearly: $\eta = \theta^T x$. (Or, if η is vector-valued, then $\eta_i = \theta_i^T x$.)

The third of these assumptions might seem the least well justified of the above, and it might be better thought of as a "design choice" in our recipe for designing GLMs, rather than as an assumption per se. These

---

[7]If we leave σ^2 as a variable, the Gaussian distribution can also be shown to be in the exponential family, where $\eta \in \mathbb{R}^2$ is now a 2-dimension vector that depends on both μ and σ. For the purposes of GLMs, however, the σ^2 parameter can also be treated by considering a more general definition of the exponential family: $p(y;\eta,\tau) = b(a,\tau) exp\left(\left(\eta^T T(y) - a(\eta)\right)/c(\tau)\right)$. Here, τ is called the **dispersion parameter**, and for the Gaussian, $c(\tau) = \sigma^2$; but given our simplification above, we won't need the more general definition for the examples we will consider here.

three assumptions/design choices will allow us to derive a very elegant class of learning algorithms, namely GLMs, that have many desirable properties such as ease of learning. Furthermore, the resulting models are often very effective for modelling different types of distributions over y; for example, we will shortly show that both logistic regression and ordinary least squares can both be derived as GLMs.

2.1.13.1 Ordinary Least Squares

To show that ordinary least squares is a special case of the GLM family of models, consider the setting where the target variable y (also called the **response variable** in GLM terminology) is continuous, and we model the conditional distribution of y given x as as a Gaussian $\mathcal{N}\left(\mu, \sigma^2\right)$. (Here, μ may depend x.) So, we let the $ExponentialFamily\left(\eta\right)$ distribution above be the Gaussian distribution. As we saw previously, in the formulation of the Gaussian as an exponential family distribution, we had $\mu = \eta$. So, we have

$$
\begin{aligned}
h_\theta\left(x\right) &= E\left[y|x;\theta\right] \\
&= \mu \\
&= \eta \\
&= \theta^T x.
\end{aligned}
\tag{2.50}
$$

The first equality follows from Assumption 2, above; the second equality follows from the fact that $y|x;\theta \sim \mathcal{N}\left(\mu, \sigma^2\right)$, and so its expected value is given by μ; the third equality follows from Assumption 1 (and our earlier derivation showing that $\mu = \eta$ in the formulation of the Gaussian as an exponential family distribution); and the last equality follows from Assumption 3.

2.1.13.2 Logistic Regression

We now consider logistic regression. Here we are interested in binary classification, so $y \in \{0, 1\}$. Given that y is binary-valued, it therefore seems natural to choose the Bernoulli family of distributions to model the conditional distribution of y given x. In our formulation of the Bernoulli distribution as an exponential family distribution, we had $\Phi = 1/\left(1 + e^{-\eta}\right)$. Furthermore, note that if $y|x;\theta \sim \text{Bernoulli}\left(\Phi\right)$, then $E\left[y|x;\theta\right] = \Phi$. So, following a similar derivation as the one for ordinary least squares, we get:

$$
\begin{aligned}
h_\theta\left(x\right) &= E\left[y|x;\theta\right] \\
&= \Phi \\
&= 1/\left(1 + e^{-\eta}\right) \\
&= 1/\left(1 + e^{-\theta^T x}\right)
\end{aligned}
\tag{2.51}
$$

So, this gives us hypothesis functions of the form $h_\theta\left(x\right) = 1/\left(1 + e^{-\theta^T x}\right)$. If you are previously wondering how we came up with the form of the logistic function $1/\left(1 + e^{-z}\right)$, this gives one answer: Once we assume that y conditioned on x is Bernoulli, it arises as a consequence of the definition of GLMs and exponential family distributions.

To introduce a little more terminology, the function g giving the distribution's mean as a function of the natural parameter $\left(g\left(\eta\right) = E\left[T\left(y\right);\eta\right]\right)$ is called the **canonical response function**. Its inverse, g^{-1}, is called the **canonical link function**. Thus, the canonical response function for the Gaussian family is just the identify function; and the canonical response function for the Bernoulli is the logistic function.[8]

---

[8]Many texts use g to denote the link function, and g^{-1} to denote the response function; but the notation we're using here, inherited from the early machine learning literature, will be more consistent with the notation used in the rest of the class.

2.1.13.3 Softmax Regression

Let's look at one more example of a GLM. Consider a classification problem in which the response variable y can take on any one of k values, so $y \in \{1, 2, ..., k\}$. For example, rather than classifying email into the two classes spam or not-spam—which would have been a binary classification problem—we might want to classify it into three classes, such as spam, personal mail, and work-related mail. The response variable is still discrete, but can now take on more than two values. We will thus model it as distributed according to a multinomial distribution.

Let's derive a GLM for modelling this type of multinomial data. To do so, we will begin by expressing the multinomial as an exponential family distribution.

To parameterize a multinomial over k possible outcomes, one could use k parameters $\Phi_1, ..., \Phi_k$ specifying the probability of each of the outcomes. However, these parameters would be redundant, or more formally, they would not be independent (since knowing any $k-1$ of the Φ_i's uniquely determines the last one, as they must satisfy $\sum_{i=1}^{k} \Phi_i = 1$). So, we will instead parameterize the multinomial with only $k-1$ parameters, $\Phi_1, ..., \Phi_{k-1}$, where $\Phi_i = p(y = i; \Phi)$, and $p(y = k; \Phi) = 1 - \sum_{i=1}^{k-1} \Phi_i$. For notational convenience, we will also let $\Phi_k = 1 - \sum_{i=1}^{k-1} \Phi_i$, but we should keep in mind that this is not a parameter, and that it is fully specified by $\Phi_1, ..., \Phi_{k-1}$.

To express the multinomial as an exponential family distribution, we will define $T(y) \in \mathbb{R}^{k-1}$ as follows:

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, T(2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, T(3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots, T(k-1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, T(k) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad (2.52)$$

Unlike our previous examples, here we do *not* have $T(y) = y$; also, $T(y)$ is now a $k-1$ dimensional vector, rather than a real number. We will write $(T(y))_i$ to denote the i-th element of the vector $T(y)$.

We introduce one more very useful piece of notation. An indicator function $1\{\cdot\}$ takes on a value of 1 if its argument is true, and 0 otherwise ($1\{\text{True}\} = 1$, $1\{\text{False}\} = 0$). For example, $1\{2 = 3\} = 0$, and $1\{3 = 5 - 2\} = 1$. So, we can also write the relationship between $T(y)$ and y as $(T(y))_i = 1\{y = i\}$. (Before you continue reading, please make sure you understand why this is true!) Further, we have that $E[(T(y))_i] = P(y = i) = \Phi_i$.

We are now ready to show that the multinomial is a member of the exponential family. We have:

$$\begin{aligned} p(y; \Phi) &= \Phi_1^{1\{y=1\}} \Phi_2^{1\{y=2\}} \cdots \Phi_k^{1\{y=k\}} \\ &= \Phi_1^{1\{y=1\}} \Phi_2^{1\{y=2\}} \cdots \Phi_k^{1 - \sum_{i=1}^{k-1} 1\{y=i\}} \\ &= \Phi_1^{(T(y))_1} \Phi_2^{(T(y))_2} \cdots \Phi_k^{1 - \sum_{i=1}^{k-1} (T(y))_i} \\ &= exp\left((T(y))_1 log(\Phi_1) + (T(y))_2 log(\Phi_2) + \right. \\ &\qquad \left. \cdots + \left(1 - \sum_{i=1}^{k-1} (T(y))_i\right) log(\Phi_k) \right) \\ &= exp\left((T(y))_1 log(\Phi_1/\Phi_k) + (T(y))_2 log(\Phi_2/\Phi_k) + \right. \\ &\qquad \left. \cdots + (T(y))_{k-1} log(\Phi_{k-1}/\Phi_k) + log(\Phi_k) \right) \\ &= b(y) exp\left(\eta^T T(y) - a(\eta)\right) \end{aligned} \qquad (2.53)$$

where

$$\eta \ = \ \begin{bmatrix} log\,(\Phi_1/\Phi_k) \\ log\,(\Phi_2/\Phi_k) \\ \vdots \\ log\,(\Phi_{k-1}/\Phi_k) \end{bmatrix},$$

$$a\,(\eta) \ = \ -log\,(\Phi_k)$$

$$b\,(y) \ = \ 1. \tag{2.54}$$

This completes our formulation of the multinomial as an exponential family distribution.

The link function is given (for $i = 1, ..., k$) by

$$\eta_i = log\frac{\Phi_i}{\Phi_k}. \tag{2.55}$$

For convenience, we have also defined $\eta_k = log\,(\Phi_k/\Phi_k) = 0$. To invert the link function and derive the response function, we therefore have that

$$e^{\eta_i} \ = \ \frac{\Phi_i}{\Phi_k}$$

$$\Phi_k e^{\eta_i} \ = \ \Phi_i$$

$$\Phi_k \sum_{i=1}^{k} e^{\eta_i} \ = \ \sum_{i=1}^{k} \Phi_i = 1 \tag{2.56}$$

This implies that $\Phi_k = 1/\sum_{i=1}^{k} e^{\eta_i}$, which can be substituted back into Equation (2.56) to give the response function

$$\Phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}} \tag{2.57}$$

This function mapping from the η's to the Φ's is called the **softmax** function.

To complete our model, we use Assumption 3, given earlier, that the η_i's are linearly related to the x's. So, have $\eta_i = \theta_i^T x$ (for $i = 1, ..., k-1$), where $\theta_1, ..., \theta_{k-1} \in \mathbb{R}^{n+1}$ are the parameters of our model. For notational convenience, we can also define $\theta_k = 0$, so that $\eta_k = \theta_k^T x = 0$, as given previously. Hence, our model assumes that the conditional distribution of y given x is given by

$$p\,(y = i|x;\theta) \ = \ \Phi_i$$

$$= \ \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}}$$

$$= \ \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} e^{\theta_j^T x}} \tag{2.58}$$

This model, which applies to classification problems where $y \in \{1, ..., k\}$, is called **softmax regression**. It is a generalization of logistic regression.

Our hypothesis will output

$$
\begin{aligned}
h_\theta(x) &= E[T(y)|x;\theta] \\
&= E\left[\left.\begin{array}{c} 1\{y=1\} \\ 1\{y=2\} \\ \vdots \\ 1\{y=k-1\} \end{array} \right| x;\theta \right] \\
&= \left[\begin{array}{c} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_{k-1} \end{array} \right] \\
&= \left[\begin{array}{c} \frac{exp\theta_1^T x}{\sum_{j=1}^{k} exp\theta_j^T x} \\ \frac{exp\theta_2^T x}{\sum_{j=1}^{k} exp\theta_j^T x} \\ \vdots \\ \frac{exp\theta_{k-1}^T x}{\sum_{j=1}^{k} exp\theta_j^T x} \end{array} \right].
\end{aligned}
\tag{2.59}
$$

In other words, our hypothesis will output the estimated probability that $p(y=i|x;\theta)$, for every value of $i = 1, ..., k$. (Even though $h_\theta(x)$ as defined above is only $k-1$ dimensional, clearly $p(y=k|x;\theta)$ can be obtained as $1 - \sum_{i=1}^{k-1} \Phi_i$.)

Lastly, let's discuss parameter fitting. Similar to our original derivation of ordinary least squares and logistic regression, if we have a training set of m examples $\{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$ and would like to learn the parameters θ_i of this model, we would begin by writing down the log-likelihood

$$
\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{m} log\, p\left(y^{(i)}|x^{(i)};\theta\right) \\
&= \sum_{i=1}^{m} log \prod_{l=1}^{k} \left(\frac{e^{\theta_l^T x^{(i)}}}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \right)^{1\{y^{(i)}=l\}}
\end{aligned}
\tag{2.60}
$$

To obtain the second line above, we used the definition for $p(y|x;\theta)$ given in Equation (2.58). We can now obtain the maximum likelihood estimate of the parameters by maximizing $\ell(\theta)$ in terms of θ, using a method such as gradient ascent or Newton's method.

2.2 Machine Learning Lecture 2 Course Notes[9]

2.2.1 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.

---

[9]This content is available online at $<$http://cnx.org/content/m45861/1.4/$>$.

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 an 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:

$$p(y|x) = \frac{p(x|y)\, p(y)}{p(x)}.$$ (2.61)

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

$$
\begin{aligned}
\underset{y}{argmax}\, p(y|x) &= \underset{y}{argmax}\, \frac{p(x|y)p(y)}{p(x)} \\
&= \underset{y}{argmax}\, p(x|y)\, p(y).
\end{aligned}
$$ (2.62)

2.2.2 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. Let's talk briefly about the properties of multivariate normal distributions before moving on to the GDA model itself.

2.2.2.1 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:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right).$$ (2.63)

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:

$$E[X] = \int_x x\, p(x; \mu, \Sigma)\, dx = \mu$$ (2.64)

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

$$\text{Cov}(X) = \Sigma.$$ (2.65)

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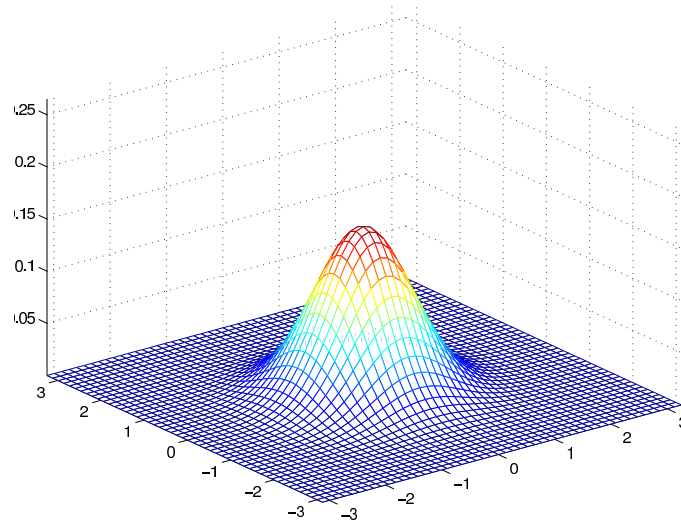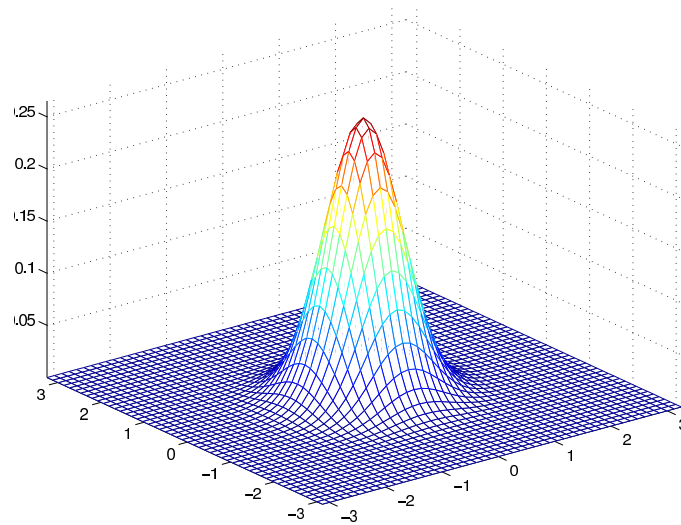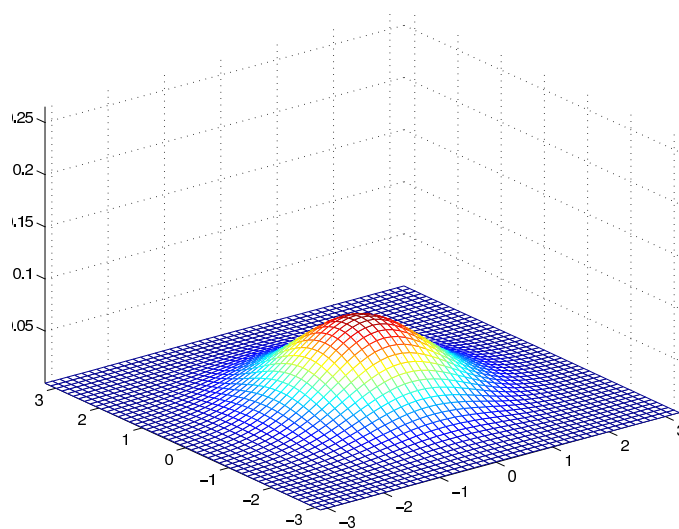

Figure 2.12

Figure 2.13

Figure 2.14

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."

Let's look at some more examples.
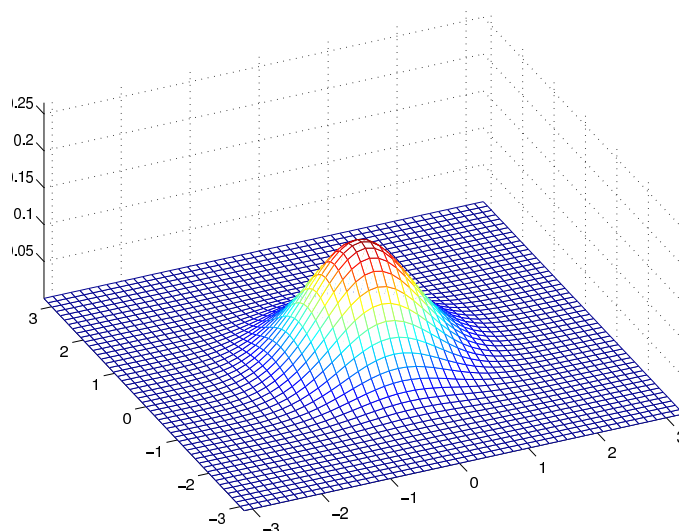
Figure 2.15
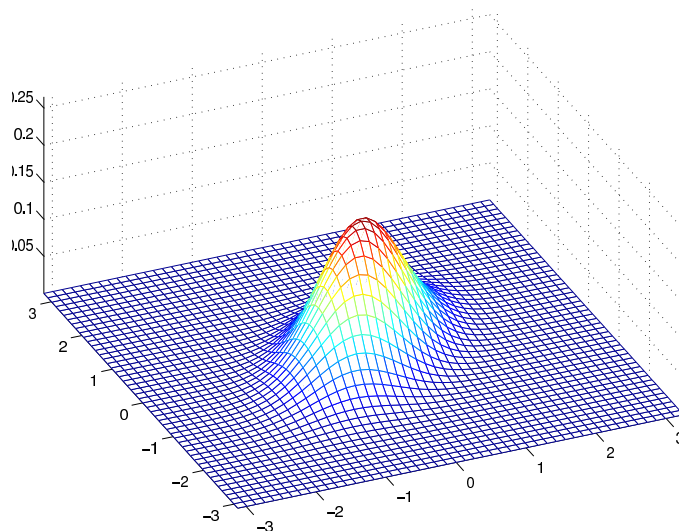
Figure 2.16
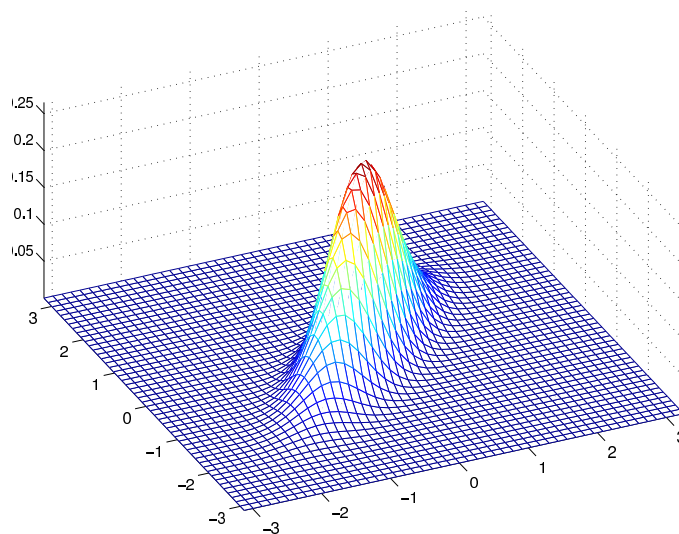
Figure 2.17

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

$$\Sigma = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right]; \quad \Sigma = \left[\begin{array}{cc} 1 & 0.5 \\ 0.5 & 1 \end{array} \right]; \quad .\Sigma = \left[\begin{array}{cc} 1 & 0.8 \\ 0.8 & 1 \end{array} \right]. \tag{2.66}$$

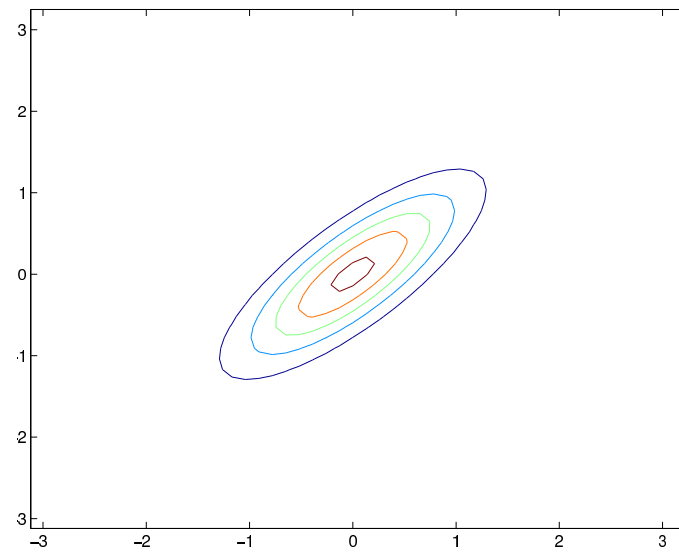The leftmost figure shows the familiar standard normal distribution, and we see that as we increase the

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:
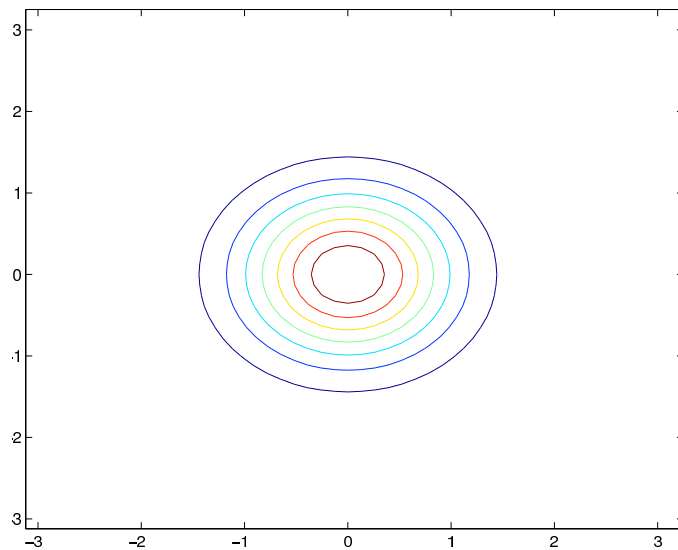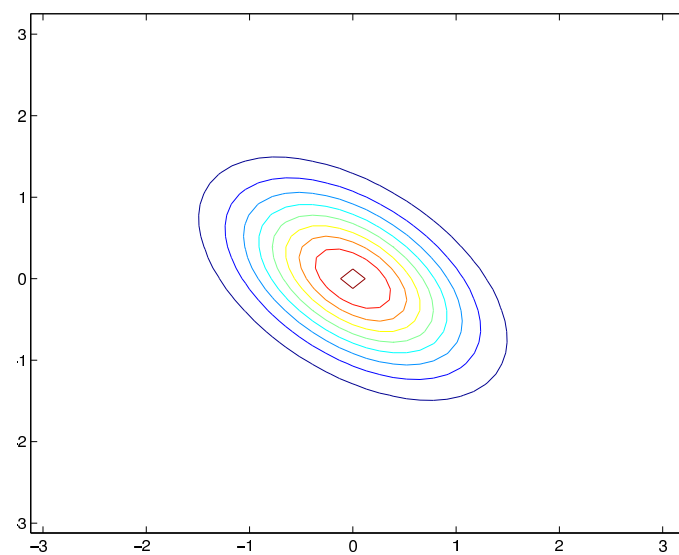

Figure 2.18
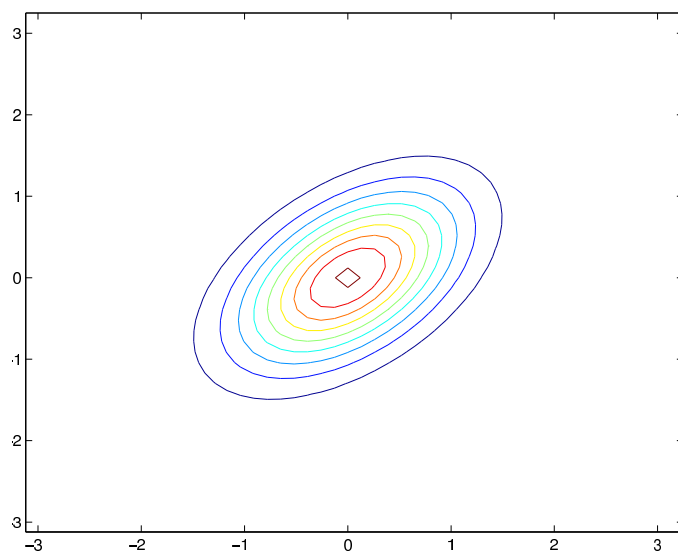
Figure 2.19

Figure 2.20

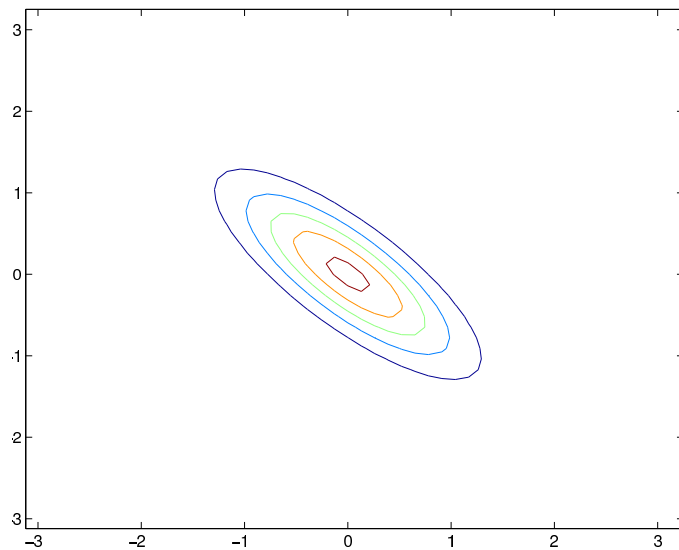Here's one last set of examples generated by varying Σ:
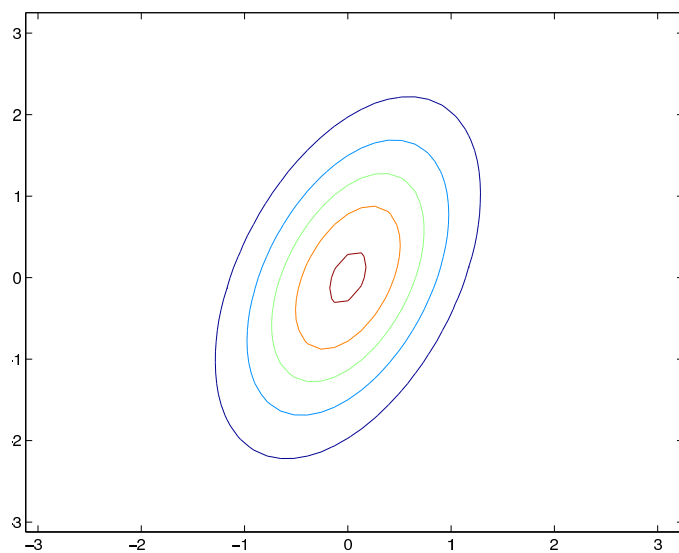

Figure 2.21

Figure 2.22

Figure 2.23

The plots above used, respectively,

$$\Sigma = \left[\begin{array}{cc} 1 & -0.5 \\ -0.5 & 1 \end{array}\right]; \;\; \Sigma = \left[\begin{array}{cc} 1 & -0.8 \\ -0.8 & 1 \end{array}\right]; \;\; .\Sigma = \left[\begin{array}{cc} 3 & 0.8 \\ 0.8 & 1 \end{array}\right]. \tag{2.67}$$

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).

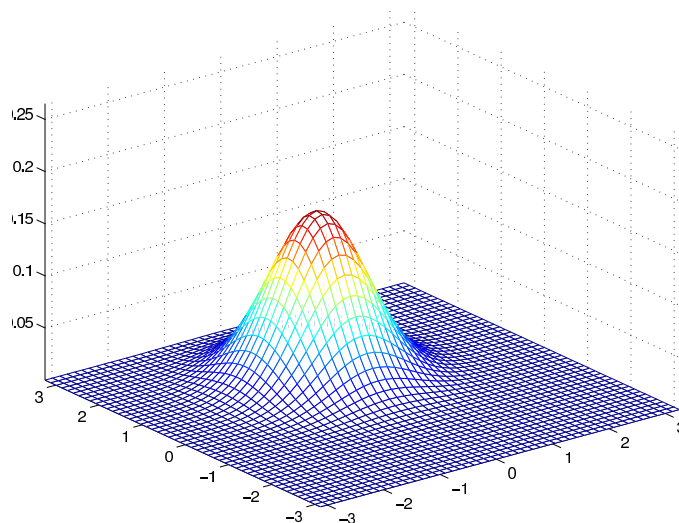As our last set of examples, fixing $\Sigma = I$, by varying μ, we can also move the mean of the density around.
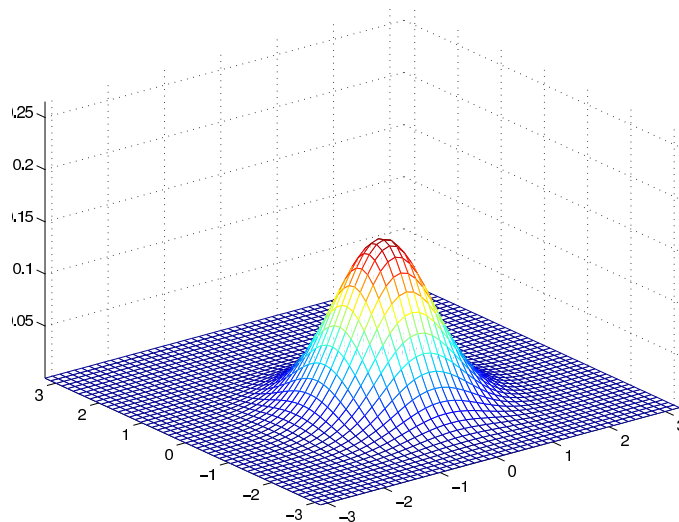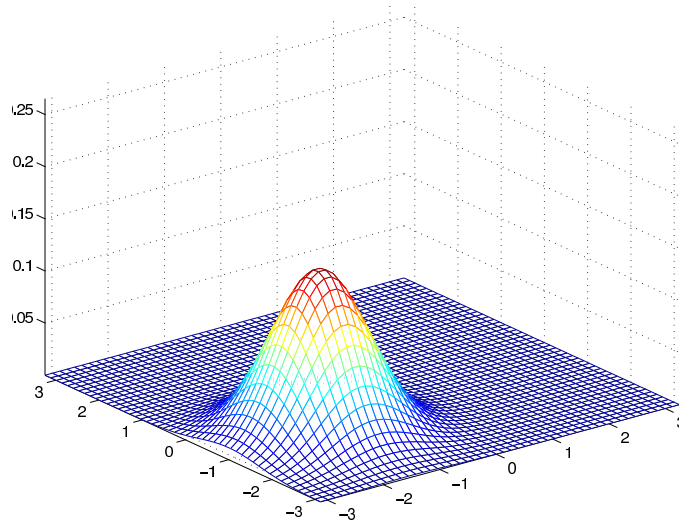

Figure 2.24

Figure 2.25

Figure 2.26

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

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix} ; \quad \mu = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} ; \quad \mu = \begin{bmatrix} -1 \\ -1.5 \end{bmatrix} . \tag{2.68}$$

2.2.2.2 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 $p\left(x|y\right)$ using a multivariate normal distribution. The model is:

$$\begin{aligned} y & \sim & \text{Bernoulli}\left(\Phi\right) \\ x|y = 0 & \sim & \mathcal{N}\left(\mu_0, \Sigma\right) \\ x|y = 1 & \sim & \mathcal{N}\left(\mu_1, \Sigma\right) \end{aligned} \tag{2.69}$$

Writing out the distributions, this is:

$$\begin{aligned} p\left(y\right) & = & \Phi^y(1-\Phi)^{1-y} \\ p\left(x|y=0\right) & = & \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}\left(x-\mu_0\right)\right) \\ p\left(x|y=1\right) & = & \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp\left(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}\left(x-\mu_1\right)\right) \end{aligned} \tag{2.70}$$

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

$$\begin{aligned} \ell\left(\Phi, \mu_0, \mu_1, \Sigma\right) & = & log \prod_{i=1}^{m} p\left(x^{(i)}, y^{(i)}; \Phi, \mu_0, \mu_1, \Sigma\right) \\ & = & log \prod_{i=1}^{m} p\left(x^{(i)}|y^{(i)}; \mu_0, \mu_1, \Sigma\right) p\left(y^{(i)}; \Phi\right). \end{aligned} \tag{2.71}$$

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

$$
\begin{aligned}
\Phi &= \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\} \\
\mu_0 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}} \\
\mu_1 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}} \\
\Sigma &= \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - \mu_{y^{(i)}}\right)\left(x^{(i)} - \mu_{y^{(i)}}\right)^{T}.
\end{aligned}
\tag{2.72}
$$

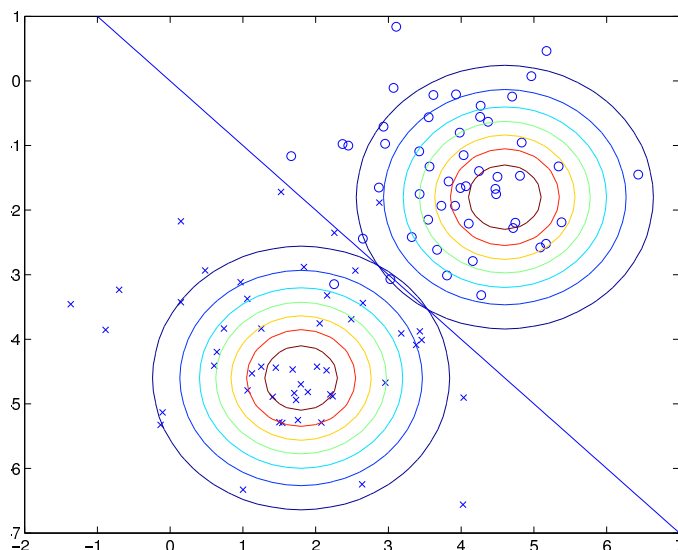Pictorially, what the algorithm is doing can be seen in as follows:

Figure 2.27

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 $p\left(y = 1|x\right) = 0.5$. 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 $y = 0$.

2.2.2.3 Discussion: GDA and logistic regression

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

$$
p\left(y = 1|x; \Phi, \Sigma, \mu_0, \mu_1\right) = \frac{1}{1 + exp\left(-\theta^T x\right)},
\tag{2.73}
$$

where θ is some appropriate function of $\Phi, \Sigma, \mu_0, \mu_1$.[10] 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 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.)

2.2.3 Naive Bayes

In GDA, the feature vectors x were continuous, real-valued vectors. Let's 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**.

Let's 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 $x_i = 1$;

---

[10]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.

otherwise, we let $x_i = 0$. For instance, the vector

$$x = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} a \\ aardvark \\ aardwolf \\ \vdots \\ buy \\ \vdots \\ zygmurgy \end{matrix} \tag{2.74}$$

is used to represent an email that contains the words "a" and "buy," but not "aardvark," "aardwolf" or "zygmurgy."[11] The set of words encoded into the 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 generative 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:

$$\begin{aligned} p(x_1, ..., x_{50000}|y) &= p(x_1|y)\,p(x_2|y,x_1)\,p(x_3|y,x_1,x_2)\cdots p(x_{50000}|y,x_1,...,x_{49999}) \\ &= p(x_1|y)\,p(x_2|y)\,p(x_3|y)\cdots p(x_{50000}|y) \\ &= \prod_{i=1}^{n} p(x_i|y) \end{aligned} \tag{2.75}$$

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 $\{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$, we can write down the joint likelihood of the data:

$$\mathcal{L}\left(\Phi_y, \Phi_{j|y=0}, \Phi_{j|y=1}\right) = \prod_{i=1}^{m} p\left(x^{(i)}, y^{(i)}\right). \tag{2.76}$$

---

[11] 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, 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.

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

$$
\begin{aligned}
\Phi_{j|y=1} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)}=1 \wedge y^{(i)}=1\}}{\sum_{i=1}^m 1\{y^{(i)}=1\}} \\
\Phi_{j|y=0} &= \frac{\sum_{i=1}^m 1\{x_j^{(i)}=1 \wedge y^{(i)}=0\}}{\sum_{i=1}^m 1\{y^{(i)}=0\}} \\
\Phi_y &= \frac{\sum_{i=1}^m 1\{y^{(i)}=1\}}{m}
\end{aligned}
\tag{2.77}
$$

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

$$
\begin{aligned}
p\,(y=1|x) &= \frac{p(x|y=1)p(y=1)}{p(x)} \\
&= \frac{\left(\prod_{i=1}^n p(x_i|y=1)\right)p(y=1)}{\left(\prod_{i=1}^n p(x_i|y=1)\right)p(y=1)+\left(\prod_{i=1}^n p(x_i|y=0)\right)p(y=0)},
\end{aligned}
\tag{2.78}
$$

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 $\{1, 2, ..., k_i\}$ 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

Table 2.3

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.

2.2.3.1 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. Let's 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 let's 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

$$
\begin{aligned}
\Phi_{35000|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)}=1 \wedge y^{(i)}=1\}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}} = 0 \\
\Phi_{35000|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)}=1 \wedge y^{(i)}=0\}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}} = 0
\end{aligned}
\tag{2.79}
$$

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

$$
\begin{aligned}
p\left(y=1|x\right) &= \frac{\prod_{i=1}^{n} p(x_i|y=1)p(y=1)}{\prod_{i=1}^{n} p(x_i|y=1)p(y=1)+\prod_{i=1}^{n} p(x_i|y=0)p(y=0)} \\
&= \frac{0}{0}.
\end{aligned}
\tag{2.80}
$$

This is because each of the terms "$\prod_{i=1}^{n} p\left(x_i|y\right)$" includes a term $p\left(x_{35000}|y\right)=0$ 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 $\{1, ..., k\}$. We can parameterize our multinomial with $\Phi_i = p\left(z=i\right)$. Given a set of m independent observations $\{z^{(1)}, ..., z^{(m)}\}$, the maximum likelihood estimates are given by

$$
\Phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)}=j\}}{m}.
\tag{2.81}
$$

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

$$
\Phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)}=j\}+1}{m+k}.
\tag{2.82}
$$

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:

$$
\begin{aligned}
\Phi_{j|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=1\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}+2} \\
\Phi_{j|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=0\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}+2}
\end{aligned}
\tag{2.83}
$$

(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 $p\left(y=1\right)$ and will be quite far from 0 anyway.)

2.2.3.2 Event models for text classification

To close off our discussion of generative learning algorithms, let's 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 **multivariate 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 $\{1, ..., |V|\}$, where $|V|$ is the size of our vocabulary (dictionary). An email of n words is now represented by a vector $(x_1, x_2, ..., x_n)$ 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, $\Phi_{k|y=1} = p(x_j = k|y = 1)$ (for any j) and $\Phi_{i|y=0} = p(x_j = k|y = 0)$. 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 $\{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$ where $x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}, ..., x_{n_i}^{(i)}\right)$ (here, n_i is the number of words in the i-training example), the likelihood of the data is given by

$$
\begin{aligned}
\mathcal{L}\left(\Phi, \Phi_{k|y=0}, \Phi_{k|y=1}\right) &= \prod_{i=1}^{m} p\left(x^{(i)}, y^{(i)}\right) \\
&= \prod_{i=1}^{m} \left(\prod_{j=1}^{n_i} p\left(x_j^{(i)}|y; \Phi_{k|y=0}, \Phi_{k|y=1}\right)\right) p\left(y^{(i)}; \Phi_y\right).
\end{aligned} \tag{2.84}
$$

Maximizing this yields the maximum likelihood estimates of the parameters:

$$
\begin{aligned}
\Phi_{k|y=1} &= \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=1\}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}n_i} \\
\Phi_{k|y=0} &= \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=0\}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}n_i} \\
\Phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}{m}.
\end{aligned} \tag{2.85}
$$

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:

$$
\begin{aligned}
\Phi_{k|y=1} &= \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=1\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}n_i+|V|} \\
\Phi_{k|y=0} &= \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=0\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}n_i+|V|}.
\end{aligned} \tag{2.86}
$$

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.

2.3 Machine Learning Lecture 3 Course Notes[12]

2.3.1 Support Vector Machines

This set of notes presents the Support Vector Machine (SVM) learning algorithm. SVMs are among the best (and many believe are 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.

2.3.2 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 "Functional and geometric margins" (Section 2.3.4: Functional and geometric margins).

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.
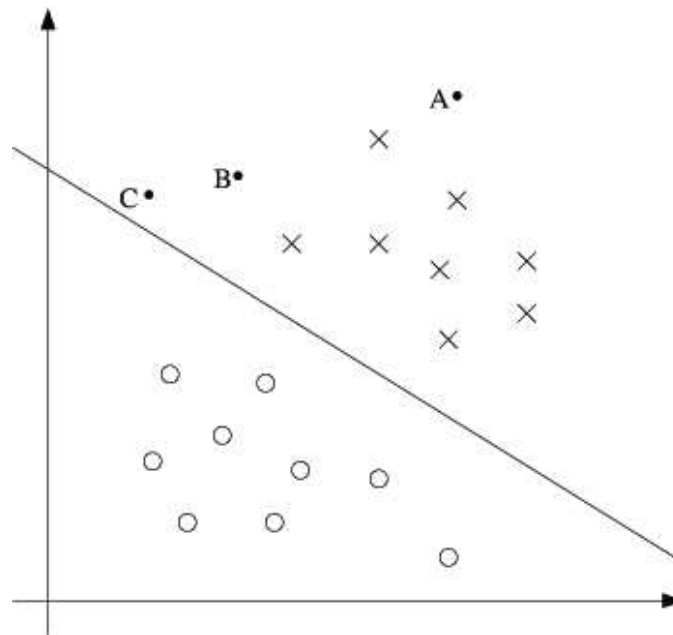

Figure 2.28

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 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 our 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.

2.3.3 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

$$h_{w,b}(x) = g\left(w^T x + b\right). \tag{2.87}$$

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 $[\theta_1 ... \theta_n]^T$.

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).

2.3.4 Functional and geometric margins

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

$$\hat{\gamma}^{(i)} = y^{(i)} \left(w^T x + b\right). \tag{2.88}$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), 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, we need $w^T x + b$ to be a large negative number. Moreover, if $y^{(i)} \left(w^T x + b\right) > 0$, 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 $g\left(w^T x + b\right) = g\left(2w^T x + 2b\right)$, 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 $S = \{\left(x^{(i)}, y^{(i)}\right) ; i = 1, ..., m\}$, we also define the function margin of (w, b) with respect to S to be the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,...,m} \hat{\gamma}^{(i)}. \tag{2.89}$$

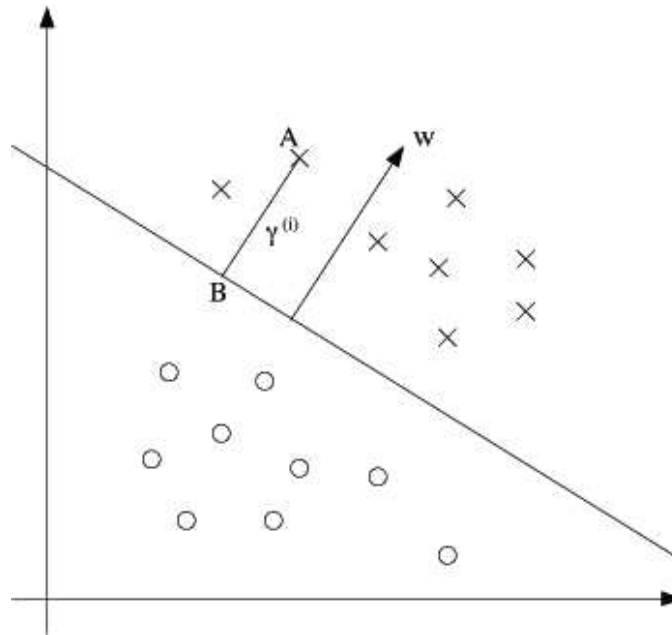Next, let's talk about **geometric margins**. Consider the picture below:

Figure 2.29

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,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0. \tag{2.90}$$

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

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}. \tag{2.91}$$

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 $\left(x^{(i)}, y^{(i)} \right)$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right). \tag{2.92}$$

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 $S = \{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$, 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:

$$\gamma = \min_{i=1,...,m} \gamma^{(i)}. \tag{2.93}$$

2.3.5 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:

$$\begin{aligned}
\max_{\gamma,w,b} \quad & \gamma \\
\text{s.t.} \quad & y^{(i)}\left(w^T x^{(i)} + b\right) \geq \gamma, \quad i = 1, ..., m \\
& ||w|| = 1.
\end{aligned} \tag{2.94}$$

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, let's try transforming the problem into a nicer one. Consider:

$$\begin{aligned}
\max_{\gamma,w,b} \quad & \frac{\hat{\gamma}}{||w||} \\
\text{s.t.} \quad & y^{(i)}\left(w^T x^{(i)} + b\right) \geq \hat{\gamma}, \quad i = 1, ..., m
\end{aligned} \tag{2.95}$$

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.

Let's 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:

$$\hat{\gamma} = 1. \tag{2.96}$$

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:

$$\min_{\gamma, w, b} \quad \frac{1}{2}||w||^2$$
$$\text{s.t.} \quad y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1, \quad i = 1, ..., m \tag{2.97}$$

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.[13]

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.

2.3.6 Lagrange duality

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

Consider a problem of the following form:

$$\min_w \quad f(w)$$
$$\text{s.t.} \quad h_i(w) = 0, \quad i = 1, ..., l. \tag{2.98}$$

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

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w) \tag{2.99}$$

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

$$\frac{\partial \mathcal{L}}{\partial w_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0, \tag{2.100}$$

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,[14] but we will give the main ideas and results, which we will then apply to our optimal margin classifier's optimization problem.

---

[13]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.

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

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

$$
\begin{aligned}
\min_w \quad & f(w) \\
\text{s.t.} \quad & g_i(w) \le 0, \quad i = 1, ..., k \\
& h_i(w) = 0, \quad i = 1, ..., l.
\end{aligned}
\tag{2.101}
$$

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

$$
\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w).
\tag{2.102}
$$

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

$$
\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \mathcal{L}(w, \alpha, \beta).
\tag{2.103}
$$

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 $h_i(w) \ne 0$ for some i), then you should be able to verify that

$$
\begin{aligned}
\theta_{\mathcal{P}}(w) &= \max_{\alpha, \beta \,:\, \alpha_i \ge 0} f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w) \\
&= \infty.
\end{aligned}
\tag{2.104}
$$

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

$$
\theta_{\mathcal{P}}(w) = \left\{ \begin{array}{ll} f(w) & \text{if } w \text{ satisfies primal constraints} \\ \infty & \text{otherwise.} \end{array} \right.
\tag{2.105}
$$

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

$$
\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \mathcal{L}(w, \alpha, \beta),
\tag{2.106}
$$

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, let's look at a slightly different problem. We define

$$
\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta).
\tag{2.107}
$$

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:

$$
\max_{\alpha, \beta \,:\, \alpha_i \ge 0} \theta_{\mathcal{D}}(\alpha, \beta) = \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \min_w \mathcal{L}(w, \alpha, \beta).
\tag{2.108}
$$

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 $d^* = \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \theta_{\mathcal{D}}(w)$.

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

$$
d^* = \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \min_w \mathcal{L}(w, \alpha, \beta) \le \min_w \max_{\alpha, \beta \,:\, \alpha_i \ge 0} \mathcal{L}(w, \alpha, \beta) = p^*.
\tag{2.109}
$$

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

$$d^* = p^*, \tag{2.110}$$

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

Suppose f and the g_i's are convex,[15] and the h_i's are affine.[16] 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.

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:

$$
\begin{aligned}
\frac{\partial}{\partial w_i} \mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \;\; i = 1, ..., n \\
\frac{\partial}{\partial \beta_i} \mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \;\; i = 1, ..., l \\
\alpha_i^* g_i(w^*) &= 0, \;\; i = 1, ..., k \\
g_i(w^*) &\leq 0, \;\; i = 1, ..., k \\
\alpha^* &\geq 0, \;\; i = 1, ..., k
\end{aligned}
\tag{2.111}
$$

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 the third equation in (2.111), 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.

2.3.7 Optimal margin classifiers

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

$$
\begin{aligned}
\min_{\gamma, w, b} \quad & \frac{1}{2}||w||^2 \\
\text{s.t.} \quad & y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1, \;\; i = 1, ..., m
\end{aligned}
\tag{2.112}
$$

We can write the constraints as

$$g_i(w) = -y^{(i)}\left(w^T x^{(i)} + b\right) + 1 \leq 0. \tag{2.113}$$

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.

---

[15]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.)

[16]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.
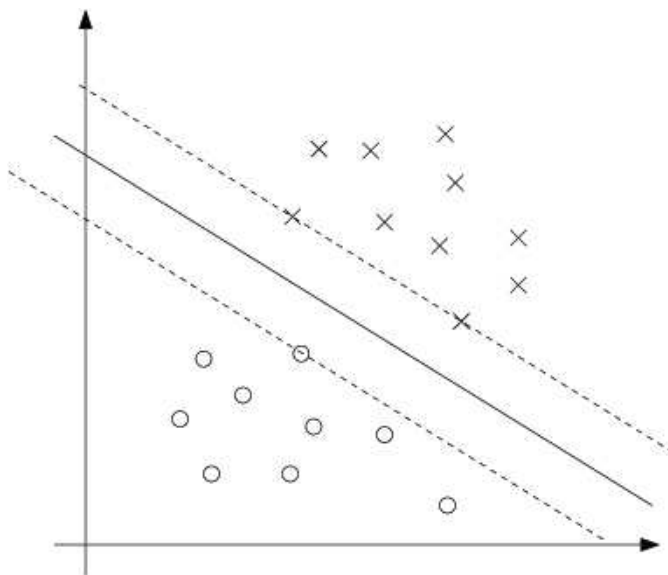
Figure 2.30

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.

Let's 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 $< x^{(i)}, x^{(j)} >$ (think of this as $\left(x^{(i)}\right)^T x^{(j)}$) 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:

$$\mathcal{L}\left(w, b, \alpha\right) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}\left(w^T x^{(i)} + b\right) - 1\right].$$ (2.114)

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

Let's find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}\left(w, b, \alpha\right)$ 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:

$$\nabla_w \mathcal{L}\left(w, b, \alpha\right) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$ (2.115)

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}. \tag{2.116}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{2.117}$$

If we take the definition of w in Equation (2.116) and plug that back into the Lagrangian (Equation (2.114)), and simplify, we get

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left(x^{(i)}\right)^T x^{(j)} - b \sum_{i=1}^{m} \alpha_i y^{(i)}. \tag{2.118}$$

But from Equation (2.117), the last term must be zero, so we obtain

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left(x^{(i)}\right)^T x^{(j)}. \tag{2.119}$$

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 (2.117), we obtain the following dual optimization problem:

$$
\begin{aligned}
\max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j <x^{(i)}, x^{(j)}> . \\
\text{s.t.} \quad & \alpha_i \geq 0, \;\; i = 1, ..., m \\
& \sum_{i=1}^{m} \alpha_i y^{(i)} = 0,
\end{aligned}
\tag{2.120}
$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions ((2.111)) 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 (2.116) 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

$$b^* = -\frac{max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}. \tag{2.121}$$

(Check for yourself that this is correct.)

Before moving on, let's also take a more careful look at Equation (2.116), 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 $w^T x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (2.116), this quantity can also be written:

$$
\begin{aligned}
w^T x + b & = \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}\right)^T x + b \\
& = \sum_{i=1}^{m} \alpha_i y^{(i)} <x^{(i)}, x> + b.
\end{aligned}
\tag{2.122}
$$

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 (2.122) 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.

2.3.8 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

$$\Phi\left(x\right) = \left[\begin{array}{c} x \\ x^2 \\ x^3 \end{array}\right]. \tag{2.123}$$

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

Since the algorithm can be written entirely in terms of the inner products $< x, z >$, this means that we would replace all those inner products with $< \Phi\left(x\right), \Phi\left(z\right) >$. Specificically, given a feature mapping Φ, we define the corresponding **Kernel** to be

$$K\left(x, z\right) = \Phi(x)^T \Phi\left(z\right). \tag{2.124}$$

Then, everywhere we previously had $< x, z >$ in our algorithm, we could simply replace it with $K\left(x, z\right)$, and our algorithm would now be learning using the features Φ.

Now, given Φ, we could easily compute $K\left(x, z\right)$ by finding $\Phi\left(x\right)$ and $\Phi\left(z\right)$ and taking their inner product. But what's more interesting is that often, $K\left(x, z\right)$ may be very inexpensive to calculate, even though $\Phi\left(x\right)$ 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 $K\left(x, z\right)$, 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 $\Phi\left(x\right)$.

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

$$K\left(x, z\right) = \left(x^T z\right)^2. \tag{2.125}$$

We can also write this as

$$\begin{array}{rcl} K\left(x, z\right) & = & \left(\sum_{i=1}^n x_i z_i\right)\left(\sum_{j=1}^n x_i z_i\right) \\ & = & \sum_{i=1}^n \sum_{j=1}^n x_i x_j z_i z_j \\ & = & \sum_{i,j=1}^n \left(x_i x_j\right)\left(z_i z_j\right) \end{array} \tag{2.126}$$

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

$$\Phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}. \tag{2.127}$$

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

$$\begin{aligned} K(x, z) &= \left(x^T z + c\right)^2 \\ &= \sum_{i,j=1}^n (x_i x_j)(z_i z_j) + \sum_{i=1}^n \left(\sqrt{2c} x_i\right)\left(\sqrt{2c} z_i\right) + c^2. \end{aligned} \tag{2.128}$$

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

$$\Phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \\ \sqrt{2c} x_1 \\ \sqrt{2c} x_2 \\ \sqrt{2c} x_3 \\ c \end{bmatrix}, \tag{2.129}$$

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

More broadly, the kernel $K(x, z) = \left(x^T z + c\right)^d$ corresponds to a feature mapping to an $\binom{n+d}{d}$ feature space, corresponding of all monomials of the form $x_{i_1} x_{i_2} ... x_{i_k}$ 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, let's 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

$$K(x,z) = exp\left(-\frac{||x-z||^2}{2\sigma^2}\right). \tag{2.130}$$

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) $\{x^{(1)}, ..., x^{(m)}\}$, and let a square, m-by-m matrix K be defined so that its (i,j)-entry is given by $K_{ij} = K\left(x^{(i)}, x^{(j)}\right)$. 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 $K_{ij} = K\left(x^{(i)}, x^{(j)}\right) = \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(j)}\right) = \Phi\left(x^{(j)}\right)^T \Phi\left(x^{(i)}\right) = K\left(x^{(j)}, x^{(i)}\right) = K_{ji}$, 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

$$
\begin{aligned}
z^T K z &= \sum_i \sum_j z_i K_{ij} z_j \\
&= \sum_i \sum_j z_i \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(j)}\right) z_j \\
&= \sum_i \sum_j z_i \sum_k \Phi_k\left(x^{(i)}\right) \Phi_k\left(x^{(j)}\right) z_j \\
&= \sum_k \sum_i \sum_j z_i \Phi_k\left(x^{(i)}\right) \Phi_k\left(x^{(j)}\right) z_j \\
&= \sum_k \left(\sum_i z_i \Phi_k\left(x^{(i)}\right)\right)^2 \\
&\geq 0.
\end{aligned}
\tag{2.131}
$$

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.[17]

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 $\{x^{(1)}, ..., x^{(m)}\}$, $(m < \infty)$, 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 $K(x,z) = \left(x^T z\right)^d$ or the Gaussian kernel, SVMs were able to obtain extremely good performance on this problem. This was particularly surprising

---

[17]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.

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 letters, then there are 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 $< x, z >$ 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."

2.3.9 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.
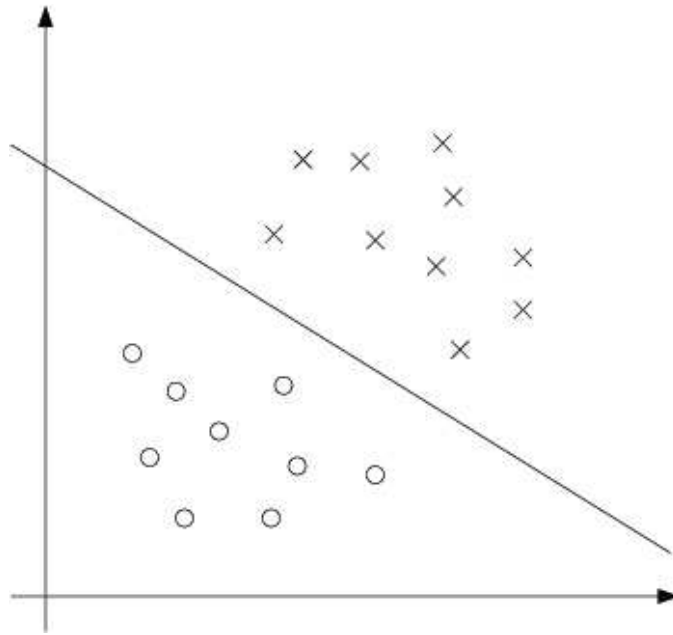
Figure 2.31
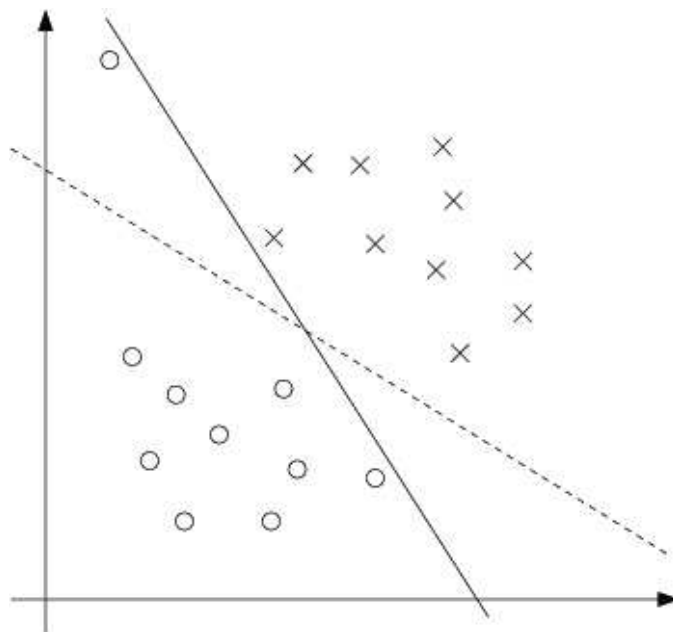
Figure 2.32

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:

$$
\begin{aligned}
\min_{\gamma,w,b} \quad & \tfrac{1}{2}||w||^2 + C\sum_{i=1}^m \xi_i \\
\text{s.t.} \quad & y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1 - \xi_i, \;\; i = 1,...,m \\
& \xi_i \geq 0, \;\; i = 1,...,m.
\end{aligned} \tag{2.132}
$$

Thus, examples are now permitted to have (functional) margin less than 1, and if an example has functional margin $1 - \xi_i$ (with $\xi > 0$), 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 $||w||^2$ small (which we saw earlier makes the margin large) and of ensuring that most examples have functional margin at least 1.

As before, we can form the Lagrangian:

$$
\mathcal{L}\left(w,b,\xi,\alpha,r\right) = \frac{1}{2}w^T w + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \alpha_i \left[y^{(i)}\left(x^T w + b\right) - 1 + \xi_i \right] - \sum_{i=1}^m r_i \xi_i. \tag{2.133}
$$

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:

$$
\begin{aligned}
\max_\alpha \quad & W\left(\alpha\right) = \sum_{i=1}^m \alpha_i - \tfrac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j < x^{(i)}, x^{(j)} > \\
\text{s.t.} \quad & 0 \leq \alpha_i \leq C, \;\; i = 1,...,m \\
& \sum_{i=1}^m \alpha_i y^{(i)} = 0,
\end{aligned} \tag{2.134}
$$

As before, we also have that w can be expressed in terms of the α_i's as given in (2.116), so that after solving the dual problem, we can continue to use (2.122) 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 ((2.121) 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:

$$
\begin{aligned}
\alpha_i = 0 \;\; &\Rightarrow \;\; y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1 \\
\alpha_i = C \;\; &\Rightarrow \;\; y^{(i)}\left(w^T x^{(i)} + b\right) \leq 1 \\
0 < \alpha_i < C \;\; &\Rightarrow \;\; y^{(i)}\left(w^T x^{(i)} + b\right) = 1.
\end{aligned} \tag{2.135}
$$

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

2.3.10 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, let's first take another digression to talk about the coordinate ascent algorithm.

2.3.10.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

$$
\max_\alpha W\left(\alpha_1, \alpha_2, ..., \alpha_m\right). \tag{2.136}
$$

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**:

1. Loop until convergence: {
2.   a. For $i = 1, ..., m$, {
        1. $\alpha_i := argmax_{\hat{\alpha}_i} W\left(\alpha_1, ..., \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, ..., \alpha_m\right)$.
    b. }
3. }

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 $\alpha_1, \alpha_2, ..., \alpha_m, \alpha_1, \alpha_2, ...$ (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 "$argmax$" 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:
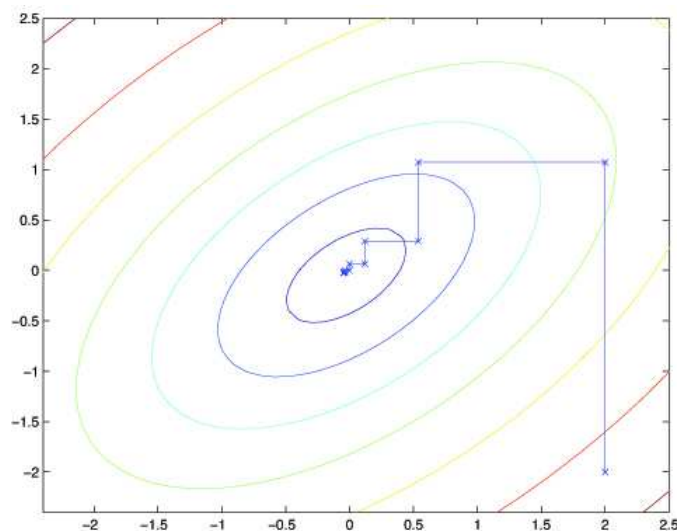

Figure 2.33

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.

2.3.10.2 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:

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

$$\text{s.t.} \qquad 0 \le \alpha_i \le C, \ \ i = 1, ..., m \tag{2.137}$$

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

Let's say we have set of α_i's that satisfy the constraints the second two equations in (2.137). Now, suppose we want to hold $\alpha_2, ..., \alpha_m$ 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 (last equation in (2.137)) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^m \alpha_i y^{(i)}. \tag{2.138}$$

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

$$\alpha_1 = -y^{(1)} \sum_{i=2}^m \alpha_i y^{(i)}. \tag{2.139}$$

(This step used the fact that $y^{(1)} \in \{-1, 1\}$, and hence $\left(y^{(1)}\right)^2 = 1$.) Hence, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, ..., \alpha_m$ fixed, then we can't make any change to α_1 without violating the constraint (last equation in (2.137)) 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 {
    a. Select some pair α_i and α_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).
    b. Reoptimize $W(\alpha)$ with respect to α_i and α_j, while holding all the other α_k's ($k \ne i, j$) fixed.

- }

To test for convergence of this algorithm, we can check whether the KKT conditions ((2.135)) 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. Let's now briefly sketch the main ideas for deriving the efficient update.

Let's say we currently have some setting of the α_i's that satisfy the constraints in (2.137), and suppose we've decided to hold $\alpha_3, ..., \alpha_m$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, ..., \alpha_m)$ with respect to α_1 and α_2 (subject to the constraints). From the final equation in (2.137), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^m \alpha_i y^{(i)}. \tag{2.140}$$

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

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \tag{2.141}$$

We can thus picture the constraints on α_1 and α_2 as follows:
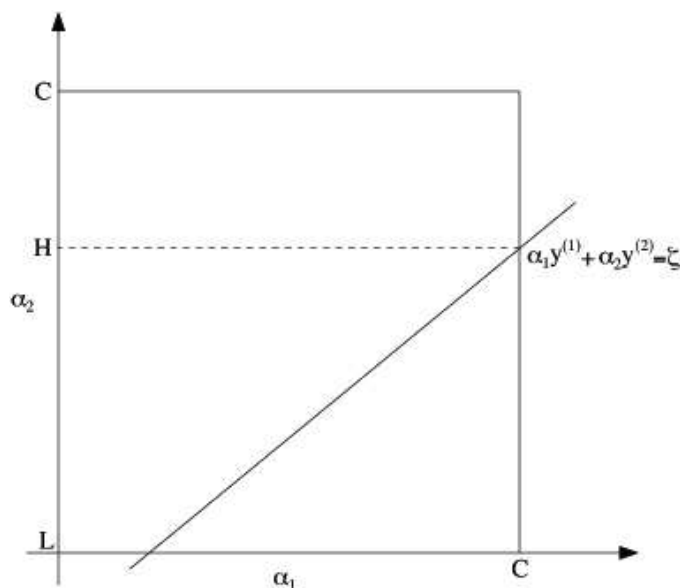
Figure 2.34

From the constraints (2.137), 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 (2.141), we can also write α_1 as a function of α_2:

$$\alpha_1 = \left(\zeta - \alpha_2 y^{(2)} \right) y^{(1)}. \tag{2.142}$$

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

$$W(\alpha_1, \alpha_2, ..., \alpha_m) = W \left(\left(\zeta - \alpha_2 y^{(2)} \right) y^{(1)}, \alpha_2, ..., \alpha_m \right). \tag{2.143}$$

Treating $\alpha_3, ..., \alpha_m$ 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 (2.137) (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

$$\alpha_2^{new} \;=\; \left\{ \begin{array}{ll} H & \text{if } \alpha_2^{\text{new, unclipped}} > H \\ \alpha_2^{\text{new, unclipped}} & L \leq \text{if } \alpha_2^{\text{new, unclipped}} \leq H \\ L & \text{if } \alpha_2^{\text{new, unclipped}} < L \end{array} \right. \tag{2.144}$$

Finally, having found the α_2^{new}, we can use Equation (2.141) 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.

2.4 Machine Learning Lecture 4 Course Notes[18]

2.4.1 Learning Theory

2.4.2 Bias/variance tradeoff

When talking about linear regression, we discussed the problem of whether to fit a "simple" model such as the linear "$y = \theta_0 + \theta_1 x$," or a more "complex" model such as the polynomial "$y = \theta_0 + \theta_1 x + \cdots \theta_5 x^5$." We saw the following example:
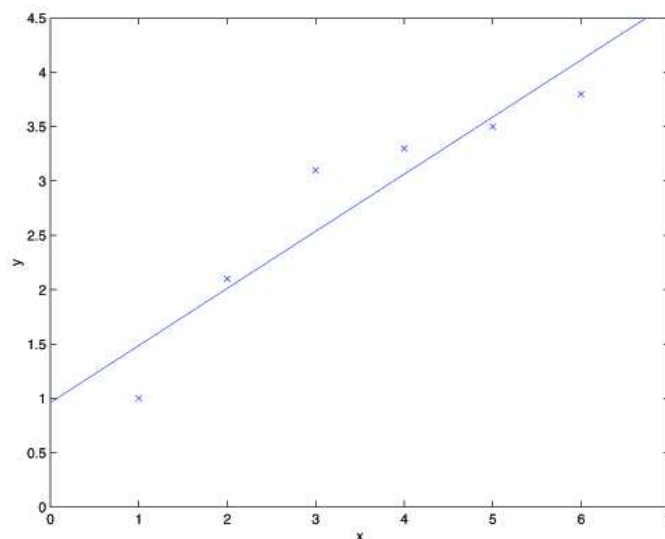

Figure 2.35

---

[18]This content is available online at <http://cnx.org/content/m45872/1.4/>.
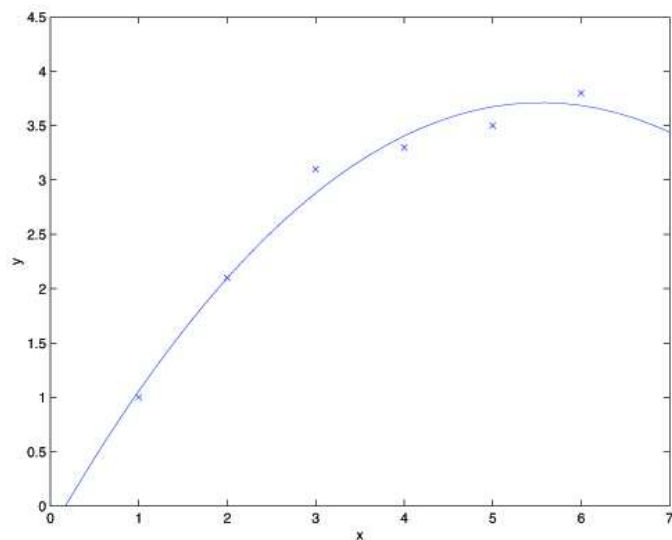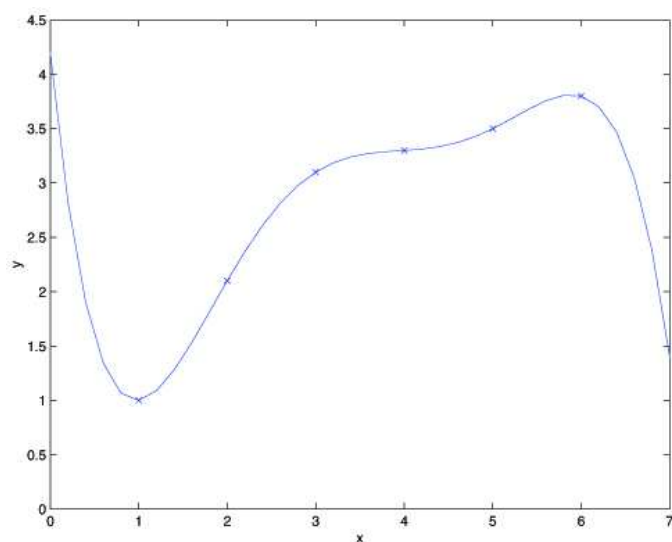
Figure 2.36

Figure 2.37

Fitting a 5th order polynomial to the data (rightmost figure) did not result in a good model. Specifically, even though the 5th order polynomial did a very good job predicting y (say, prices of houses) from x (say, living area) for the examples in the training set, we do not expect the model shown to be a good one for predicting the prices of houses not in the training set. In other words, what's has been learned from the training set does not *generalize* well to other houses. The **generalization error** (which will be made formal shortly) of a hypothesis is its expected error on examples not necessarily in the training set.

Both the models in the leftmost and the rightmost figures above have large generalization error. However, the problems that the two models suffer from are very different. If the relationship between y and x is not linear, then even if we were fitting a linear model to a very large amount of training data, the linear model would still fail to accurately capture the structure in the data. Informally, we define the **bias** of a model to be the expected generalization error even if we were to fit it to a very (say, infinitely) large training set. Thus, for the problem above, the linear model suffers from large bias, and may underfit (i.e., fail to capture structure exhibited by) the data.

Apart from bias, there's a second component to the generalization error, consisting of the **variance** of a model fitting procedure. Specifically, when fitting a 5th order polynomial as in the rightmost figure, there is a large risk that we're fitting patterns in the data that happened to be present in our small, finite training set, but that do not reflect the wider pattern of the relationship between x and y. This could be, say, because in the training set we just happened by chance to get a slightly more-expensive-than-average house here, and a slightly less-expensive-than-average house there, and so on. By fitting these "spurious" patterns in the training set, we might again obtain a model with large generalization error. In this case, we say the model has large variance.[19]

Often, there is a tradeoff between bias and variance. If our model is too "simple" and has very few parameters, then it may have large bias (but small variance); if it is too "complex" and has very many parameters, then it may suffer from large variance (but have smaller bias). In the example above, fitting a quadratic function does better than either of the extremes of a first or a fifth order polynomial.

2.4.3 Preliminaries

In this set of notes, we begin our foray into learning theory. Apart from being interesting and enlightening in its own right, this discussion will also help us hone our intuitions and derive rules of thumb about how to best apply learning algorithms in different settings. We will also seek to answer a few questions: First, can we make formal the bias/variance tradeoff that was just discussed? The will also eventually lead us to talk about model selection methods, which can, for instance, automatically decide what order polynomial to fit to a training set. Second, in machine learning it's really generalization error that we care about, but most learning algorithms fit their models to the training set. Why should doing well on the training set tell us anything about generalization error? Specifically, can we relate error on the training set to generalization error? Third and finally, are there conditions under which we can actually prove that learning algorithms will work well?

We start with two simple but very useful lemmas.

Lemma. (The union bound). Let $A_1, A_2, ..., A_k$ be k different events (that may not be independent). Then

$$P(A_1 \cup \cdots \cup A_k) \leq P(A_1) + ... + P(A_k). \tag{2.145}$$

In probability theory, the union bound is usually stated as an axiom (and thus we won't try to prove it), but it also makes intuitive sense: The probability of any one of k events happening is at most the sums of the probabilities of the k different events.

Lemma. (Hoeffding inequality) Let $Z_1, ..., Z_m$ be m independent and identically distributed (iid) random variables drawn from a Bernoulli(Φ) distribution. I.e., $P(Z_i = 1) = \Phi$, and $P(Z_i = 0) = 1 - \Phi$. Let $\hat{\Phi} = (1/m) \sum_{i=1}^{m} Z_i$ be the mean of these random variables, and let any $\gamma > 0$ be fixed. Then

$$P\left(|\Phi - \hat{\Phi}| > \gamma\right) \leq 2exp\left(-2\gamma^2 m\right) \tag{2.146}$$

---

[19]In these notes, we will not try to formalize the definitions of bias and variance beyond this discussion. While bias and variance are straightforward to define formally for, e.g., linear regression, there have been several proposals for the definitions of bias and variance for classification, and there is as yet no agreement on what is the "right" and/or the most useful formalism.

This lemma (which in learning theory is also called the **Chernoff bound**) says that if we take $\hat{\Phi}$—the average of m Bernoulli(Φ) random variables—to be our estimate of Φ, then the probability of our being far from the true value is small, so long as m is large. Another way of saying this is that if you have a biased coin whose chance of landing on heads is Φ, then if you toss it m times and calculate the fraction of times that it came up heads, that will be a good estimate of Φ with high probability (if m is large).

Using just these two lemmas, we will be able to prove some of the deepest and most important results in learning theory.

To simplify our exposition, let's restrict our attention to binary classification in which the labels are $y \in \{0, 1\}$. Everything we'll say here generalizes to other, including regression and multi-class classification, problems.

We assume we are given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, ..., m\}$ of size m, where the training examples $(x^{(i)}, y^{(i)})$ are drawn iid from some probability distribution $\mathcal{D}$. For a hypothesis h, we define the **training error** (also called the **empirical risk** or **empirical error** in learning theory) to be

$$\hat{\epsilon}(h) = \frac{1}{m} \sum_{i=1}^{m} 1\{h\left(x^{(i)}\right) \neq y^{(i)}\}. \tag{2.147}$$

This is just the fraction of training examples that h misclassifies. When we want to make explicit the dependence of $\hat{\epsilon}(h)$ on the training set S, we may also write this a $\hat{\epsilon}_S(h)$. We also define the generalization error to be

$$\epsilon(h) = P_{(x,y)\sim\mathcal{D}}\left(h(x) \neq y\right). \tag{2.148}$$

I.e. this is the probability that, if we now draw a new example (x, y) from the distribution $\mathcal{D}$, h will misclassify it.

Note that we have assumed that the training data was drawn from the *same* distribution $\mathcal{D}$ with which we're going to evaluate our hypotheses (in the definition of generalization error). This is sometimes also referred to as one of the **PAC** assumptions.[20]

Consider the setting of linear classification, and let $h_\theta(x) = 1\{\theta^T x \geq 0\}$. What's a reasonable way of fitting the parameters θ? One approach is to try to minimize the training error, and pick

$$\hat{\theta} = \arg\min_\theta \hat{\epsilon}(h_\theta). \tag{2.149}$$

We call this process **empirical risk minimization** (ERM), and the resulting hypothesis output by the learning algorithm is $\hat{h} = h_{\hat{\theta}}$. We think of ERM as the most "basic" learning algorithm, and it will be this algorithm that we focus on in these notes. (Algorithms such as logistic regression can also be viewed as approximations to empirical risk minimization.)

In our study of learning theory, it will be useful to abstract away from the specific parameterization of hypotheses and from issues such as whether we're using a linear classifier. We define the **hypothesis class** $\mathcal{H}$ used by a learning algorithm to be the set of all classifiers considered by it. For linear classification, $\mathcal{H} = \{h_\theta : h_\theta(x) = 1\{\theta^T x \geq 0\}, \theta \in \mathbb{R}^{n+1}\}$ is thus the set of all classifiers over $\mathcal{X}$ (the domain of the inputs) where the decision boundary is linear. More broadly, if we were studying, say, neural networks, then we could let $\mathcal{H}$ be the set of all classifiers representable by some neural network architecture.

Empirical risk minimization can now be thought of as a minimization over the class of functions $\mathcal{H}$, in which the learning algorithm picks the hypothesis:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\epsilon}(h) \tag{2.150}$$

---

[20]PAC stands for "probably approximately correct," which is a framework and set of assumptions under which numerous results on learning theory were proved. Of these, the assumption of training and testing on the same distribution, and the assumption of the independently drawn training examples, were the most important.

2.4.4 The case of finite $\mathcal{H}$

Let's start by considering a learning problem in which we have a finite hypothesis class $\mathcal{H} = \{h_1, ..., h_k\}$ consisting of k hypotheses. Thus, $\mathcal{H}$ is just a set of k functions mapping from $\mathcal{X}$ to $\{0, 1\}$, and empirical risk minimization selects $\hat{h}$ to be whichever of these k functions has the smallest training error.

We would like to give guarantees on the generalization error of $\hat{h}$. Our strategy for doing so will be in two parts: First, we will show that $\hat{\epsilon}(h)$ is a reliable estimate of $\epsilon(h)$ for all h. Second, we will show that this implies an upper-bound on the generalization error of $\hat{h}$.

Take any one, fixed, $h_i \in \mathcal{H}$. Consider a Bernoulli random variable Z whose distribution is defined as follows. We're going to sample $(x, y) \sim \mathcal{D}$. Then, we set $Z = 1\{h_i(x) \neq y\}$. I.e., we're going to draw one example, and let Z indicate whether h_i misclassifies it. Similarly, we also define $Z_j = 1\{h_i(x^{(j)}) \neq y^{(j)}\}$. Since our training set was drawn iid from $\mathcal{D}$, Z and the Z_j's have the same distribution.

We see that the misclassification probability on a randomly drawn example—that is, $\epsilon(h)$—is exactly the expected value of Z (and Z_j). Moreover, the training error can be written

$$\hat{\epsilon}(h_i) = \frac{1}{m}\sum_{j=1}^{m} Z_j. \tag{2.151}$$

Thus, $\hat{\epsilon}(h_i)$ is exactly the mean of the m random variables Z_j that are drawn iid from a Bernoulli distribution with mean $\epsilon(h_i)$. Hence, we can apply the Hoeffding inequality, and obtain

$$P\left(|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma\right) \leq 2exp\left(-2\gamma^2 m\right). \tag{2.152}$$

This shows that, for our particular h_i, training error will be close to generalization error with high probability, assuming m is large. But we don't just want to guarantee that $\epsilon(h_i)$ will be close to $\hat{\epsilon}(h_i)$ (with high probability) for just only one particular h_i. We want to prove that this will be true for simultaneously for $all h \in \mathcal{H}$. To do so, let A_i denote the event that $|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma$. We've already show that, for any particular A_i, it holds true that $P(A_i) \leq 2exp\left(-2\gamma^2 m\right)$. Thus, using the union bound, we have that

$$
\begin{aligned}
P\left(\exists\, h \in \mathcal{H}.|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma\right) &= P(A_1 \cup \cdots \cup A_k) \\
&\leq \sum_{i=1}^{k} P(A_i) \\
&\leq \sum_{i=1}^{k} 2exp\left(-2\gamma^2 m\right) \\
&= 2kexp\left(-2\gamma^2 m\right)
\end{aligned}
\tag{2.153}
$$

If we subtract both sides from 1, we find that

$$
\begin{aligned}
P\left(\neg\exists\, h \in \mathcal{H}.|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma\right) &= P\left(\forall h \in \mathcal{H}.|\epsilon(h_i) - \hat{\epsilon}(h_i)| \leq \gamma\right) \\
&\geq 1 - 2kexp\left(-2\gamma^2 m\right)
\end{aligned}
\tag{2.154}
$$

(The "$\neg$" symbol means "not.") So, with probability at least $1 - 2kexp\left(-2\gamma^2 m\right)$, we have that $\epsilon(h)$ will be within γ of $\hat{\epsilon}(h)$ for all $h \in \mathcal{H}$. This is called a *uniform convergence* result, because this is a bound that holds simultaneously for all (as opposed to just one) $h \in \mathcal{H}$.

In the discussion above, what we did was, for particular values of m and γ, give a bound on the probability that for some $h \in \mathcal{H}$, $|\epsilon(h) - \hat{\epsilon}(h)| > \gamma$. There are three quantities of interest here: m, γ, and the probability of error; we can bound either one in terms of the other two.

For instance, we can ask the following question: Given γ and some $\delta > 0$, how large must m be before we can guarantee that with probability at least $1 - \delta$, training error will be within γ of generalization error? By setting $\delta = 2kexp\left(-2\gamma^2 m\right)$ and solving for m, [you should convince yourself this is the right thing to do!], we find that if

$$m \geq \frac{1}{2\gamma^2} log\frac{2k}{\delta}, \tag{2.155}$$

then with probability at least $1 - \delta$, we have that $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$. (Equivalently, this shows that the probability that $|\epsilon(h) - \hat{\epsilon}(h)| > \gamma$ for some $h \in \mathcal{H}$ is at most δ.) This bound tells us how many training examples we need in order make a guarantee. The training set size m that a certain method or algorithm requires in order to achieve a certain level of performance is also called the algorithm's **sample complexity**.

The key property of the bound above is that the number of training examples needed to make this guarantee is only *logarithmic* in k, the number of hypotheses in $\mathcal{H}$. This will be important later.

Similarly, we can also hold m and δ fixed and solve for γ in the previous equation, and show [again, convince yourself that this is right!] that with probability $1 - \delta$, we have that for all $h \in \mathcal{H}$,

$$|\hat{\epsilon}(h) - \epsilon(h)| \leq \sqrt{\frac{1}{2m} log\frac{2k}{\delta}}. \tag{2.156}$$

Now, let's assume that uniform convergence holds, i.e., that $|\epsilon(h) - \hat{\epsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$. What can we prove about the generalization of our learning algorithm that picked $\hat{h} = argmin_{h \in \mathcal{H}} \hat{\epsilon}(h)$?

Define $h^* = argmin_{h \in \mathcal{H}} \epsilon(h)$ to be the best possible hypothesis in $\mathcal{H}$. Note that h^* is the best that we could possibly do given that we are using $\mathcal{H}$, so it makes sense to compare our performance to that of h^*. We have:

$$\begin{aligned}
\epsilon\left(\hat{h}\right) &\leq \hat{\epsilon}\left(\hat{h}\right) + \gamma \\
&\leq \hat{\epsilon}(h^*) + \gamma \\
&\leq \epsilon(h^*) + 2\gamma
\end{aligned} \tag{2.157}$$

The first line used the fact that $|\epsilon\left(\hat{h}\right) - \hat{\epsilon}\left(\hat{h}\right)| \leq \gamma$ (by our uniform convergence assumption). The second used the fact that $\hat{h}$ was chosen to minimize $\hat{\epsilon}(h)$, and hence $\hat{\epsilon}\left(\hat{h}\right) \leq \hat{\epsilon}(h)$ for all h, and in particular $\hat{\epsilon}\left(\hat{h}\right) \leq \hat{\epsilon}(h^*)$. The third line used the uniform convergence assumption again, to show that $\hat{\epsilon}(h^*) \leq \epsilon(h^*) + \gamma$. So, what we've shown is the following: If uniform convergence occurs, then the generalization error of $\hat{h}$ is at most 2γ worse than the best possible hypothesis in $\mathcal{H}$!

Let's put all this together into a theorem.

Theorem. Let $|\mathcal{H}| = k$, and let any m, δ be fixed. Then with probability at least $1 - \delta$, we have that

$$\epsilon\left(\hat{h}\right) \leq \left(\min_{h \in \mathcal{H}} \epsilon\left(h\right)\right) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}. \tag{2.158}$$

This is proved by letting γ equal the $\sqrt{\cdot}$ term, using our previous argument that uniform convergence occurs with probability at least $1 - \delta$, and then noting that uniform convergence implies $\epsilon\left(h\right)$ is at most 2γ higher than $\epsilon\left(h^*\right) = \min_{h \in \mathcal{H}} \epsilon\left(h\right)$ (as we showed previously).

This also quantifies what we were saying previously saying about the bias/variance tradeoff in model selection. Specifically, suppose we have some hypothesis class $\mathcal{H}$, and are considering switching to some much larger hypothesis class $\mathcal{H}' \supseteq \mathcal{H}$. If we switch to $\mathcal{H}'$, then the first term $\min_h \epsilon\left(h\right)$ can only decrease (since we'd then be taking a min over a larger set of functions). Hence, by learning using a larger hypothesis class, our "bias" can only decrease. However, if k increases, then the second $2\sqrt{\cdot}$ term would also increase. This increase corresponds to our "variance" increasing when we use a larger hypothesis class.

By holding γ and δ fixed and solving for m like we did before, we can also obtain the following sample complexity bound:

Corollary. Let $|\mathcal{H}| = k$, and let any δ, γ be fixed. Then for $\epsilon\left(\hat{h}\right) \leq \min_{h \in \mathcal{H}} \epsilon\left(h\right) + 2\gamma$ to hold with probability at least $1 - \delta$, it suffices that

$$
\begin{aligned}
m &\geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \\
&= O\left(\frac{1}{\gamma^2} \log \frac{k}{\delta}\right),
\end{aligned} \tag{2.159}
$$

2.4.5 The case of infinite $\mathcal{H}$

We have proved some useful theorems for the case of finite hypothesis classes. But many hypothesis classes, including any parameterized by real numbers (as in linear classification) actually contain an infinite number of functions. Can we prove similar results for this setting?

Let's start by going through something that is *not* the "right" argument. *Better and more general arguments exist*, but this will be useful for honing our intuitions about the domain.

Suppose we have an $\mathcal{H}$ that is parameterized by d real numbers. Since we are using a computer to represent real numbers, and IEEE double-precision floating point (`double`'s in C) uses 64 bits to represent a floating point number, this means that our learning algorithm, assuming we're using double-precision floating point, is parameterized by $64d$ bits. Thus, our hypothesis class really consists of at most $k = 2^{64d}$ different hypotheses. From the Corollary at the end of the previous section, we therefore find that, to guarantee $\epsilon\left(\hat{h}\right) \leq \epsilon\left(h^*\right) + 2\gamma$, with to hold with probability at least $1 - \delta$, it suffices that $m \geq O\left(\frac{1}{\gamma^2} \log \frac{2^{64d}}{\delta}\right) = O\left(\frac{d}{\gamma^2} \log \frac{1}{\delta}\right) = O_{\gamma, \delta}\left(d\right)$. (The γ, δ subscripts are to indicate that the last big-O is hiding constants that may depend on γ and δ.) Thus, the number of training examples needed is at most *linear* in the parameters of the model.

The fact that we relied on 64-bit floating point makes this argument not entirely satisfying, but the conclusion is nonetheless roughly correct: If what we're going to do is try to minimize training error, then in order to learn "well" using a hypothesis class that has d parameters, generally we're going to need on the order of a linear number of training examples in d.

(At this point, it's worth noting that these results were proved for an algorithm that uses empirical risk minimization. Thus, while the linear dependence of sample complexity on d does generally hold for most discriminative learning algorithms that try to minimize training error or some approximation to training error, these conclusions do not always apply as readily to discriminative learning algorithms. Giving good theoretical guarantees on many non-ERM learning algorithms is still an area of active research.)

The other part of our previous argument that's slightly unsatisfying is that it relies on the parameterization of $\mathcal{H}$. Intuitively, this doesn't seem like it should matter: We had written the class of linear classifiers as $h_\theta(x) = 1\{\theta_0 + \theta_1 x_1 + \cdots \theta_n x_n \geq 0\}$, with $n+1$ parameters $\theta_0, ..., \theta_n$. But it could also be written $h_{u,v}(x) = 1\{(u_0^2 - v_0^2) + (u_1^2 - v_1^2) x_1 + \cdots (u_n^2 - v_n^2) x_n \geq 0\}$ with $2n+2$ parameters u_i, v_i. Yet, both of these are just defining the same $\mathcal{H}$: The set of linear classifiers in n dimensions.

To derive a more satisfying argument, let's define a few more things.

Given a set $S = \{x^{(i)}, ..., x^{(d)}\}$ (no relation to the training set) of points $x^{(i)} \in \mathcal{X}$, we say that $\mathcal{H}$**shatters**S if $\mathcal{H}$ can realize any labeling on S. I.e., if for any set of labels $\{y^{(1)}, ..., y^{(d)}\}$, there exists some $h \in \mathcal{H}$ so that $h(x^{(i)}) = y^{(i)}$ for all $i = 1, ...d$.

Given a hypothesis class $\mathcal{H}$, we then define its **Vapnik-Chervonenkis dimension**, written $\text{VC}(\mathcal{H})$, to be the size of the largest set that is shattered by $\mathcal{H}$. (If $\mathcal{H}$ can shatter arbitrarily large sets, then $\text{VC}(\mathcal{H}) = \infty$.)

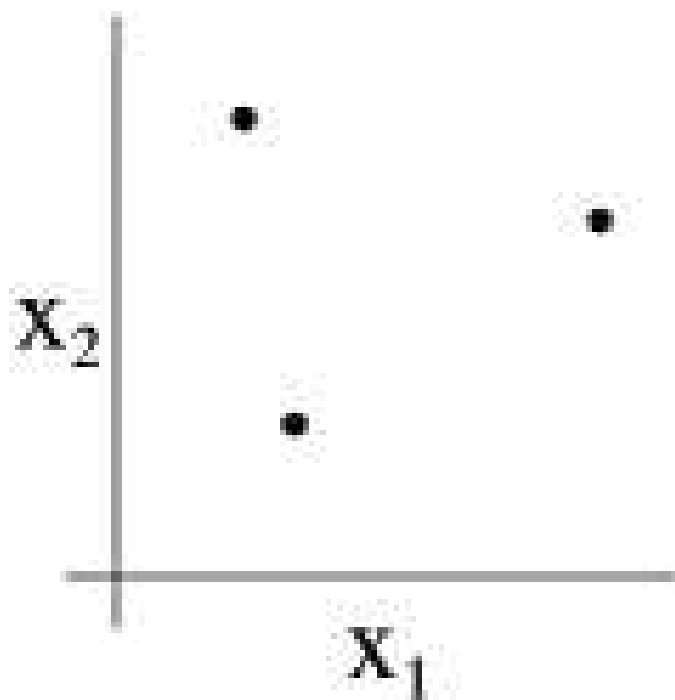For instance, consider the following set of three points:

Figure 2.38

Can the set $\mathcal{H}$ of linear classifiers in two dimensions ($h(x) = 1\{\theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0\}$) can shatter the set above? The answer is yes. Specifically, we see that, for any of the eight possible labelings of these points, we can find a linear classifier that obtains "zero training error" on them:
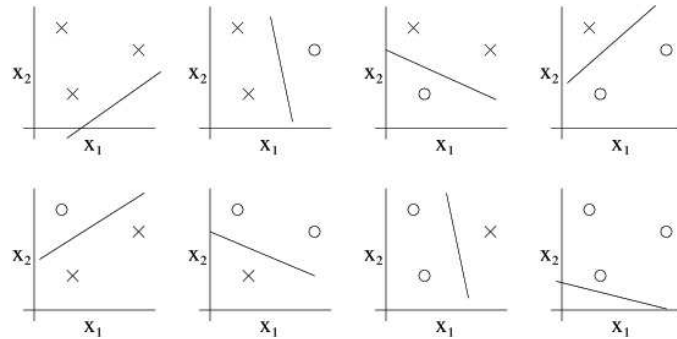
Figure 2.39

Moreover, it is possible to show that there is no set of 4 points that this hypothesis class can shatter. Thus, the largest set that $\mathcal{H}$ can shatter is of size 3, and hence $\mathrm{VC}\,(\mathcal{H}) = 3$.

Note that the VC dimension of $\mathcal{H}$ here is 3 even though there may be sets of size 3 that it cannot shatter. For instance, if we had a set of three points lying in a straight line (left figure), then there is no way to find a linear separator for the labeling of the three points shown below (right figure):
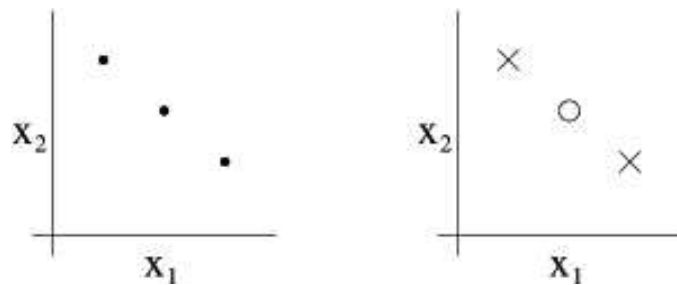
Figure 2.40

In order words, under the definition of the VC dimension, in order to prove that $\mathrm{VC}\,(\mathcal{H})$ is at least d, we need to show only that there's at least *one* set of size d that $\mathcal{H}$ can shatter.

The following theorem, due to Vapnik, can then be shown. (This is, many would argue, the most important theorem in all of learning theory.)

Theorem. Let $\mathcal{H}$ be given, and let $d = \mathrm{VC}\,(\mathcal{H})$. Then with probability at least $1 - \delta$, we have that for all $h \in \mathcal{H}$,

$$|\epsilon\,(h) - \hat{\epsilon}\,(h)| \leq O\left(\sqrt{\frac{d}{m} log\frac{m}{d} + \frac{1}{m} log\frac{1}{\delta}} \right). \tag{2.160}$$

Thus, with probability at least $1 - \delta$, we also have that:

$$\epsilon\left(\hat{h}\right) \leq \epsilon\left(h^*\right) + O\left(\sqrt{\frac{d}{m}log\frac{m}{d} + \frac{1}{m}log\frac{1}{\delta}}\right). \tag{2.161}$$

In other words, if a hypothesis class has finite VC dimension, then uniform convergence occurs as m becomes large. As before, this allows us to give a bound on $\epsilon\left(h\right)$ in terms of $\epsilon\left(h^*\right)$. We also have the following corollary:

Corollary. For $|\epsilon\left(h\right) - \overset{\wedge}{\epsilon}\left(h\right)| \leq \gamma$ to hold for all $h \in \mathcal{H}$ (and hence $\epsilon\left(\hat{h}\right) \leq \epsilon\left(h^*\right) + 2\gamma$) with probability at least $1 - \delta$, it suffices that $m = O_{\gamma,\delta}\left(d\right)$.

In other words, the number of training examples needed to learn "well" using $\mathcal{H}$ is linear in the VC dimension of $\mathcal{H}$. It turns out that, for "most" hypothesis classes, the VC dimension (assuming a "reasonable" parameterization) is also roughly linear in the number of parameters. Putting these together, we conclude that (for an algorithm that tries to minimize training error) the number of training examples needed is usually roughly linear in the number of parameters of $\mathcal{H}$.

2.5 Machine Learning Lecture 5 Course Notes[21]

2.5.1 Regularization and model selection

Suppose we are trying to select among several different models for a learning problem. For instance, we might be using a polynomial regression model $h_\theta\left(x\right) = g\left(\theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_k x^k\right)$, and wish to decide if k should be 0, 1, ..., or 10. How can we automatically select a model that represents a good tradeoff between the twin evils of bias and variance[22]? Alternatively, suppose we want to automatically choose the bandwidth parameter τ for locally weighted regression, or the parameter C for our ℓ_1-regularized SVM. How can we do that?

For the sake of concreteness, in these notes we assume we have some finite set of models $\mathcal{M} = \{M_1, ..., M_d\}$ that we're trying to select among. For instance, in our first example above, the model M_i would be an i-th order polynomial regression model. (The generalization to infinite $\mathcal{M}$ is not hard.[23]) Alternatively, if we are trying to decide between using an SVM, a neural network or logistic regression, then $\mathcal{M}$ may contain these models.

2.5.2 Cross validation

Let's suppose we are, as usual, given a training set S. Given what we know about empirical risk minimization, here's what might initially seem like a algorithm, resulting from using empirical risk minimization for model selection:

1. Train each model M_i on S, to get some hypothesis h_i.
2. Pick the hypotheses with the smallest training error.

This algorithm does *not* work. Consider choosing the order of a polynomial. The higher the order of the polynomial, the better it will fit the training set S, and thus the lower the training error. Hence, this method will always select a high-variance, high-degree polynomial model, which we saw previously is often poor choice.

Here's an algorithm that works better. In **hold-out cross validation** (also called **simple cross validation**), we do the following:

1. Randomly split S into S_{train} (say, 70% of the data) and S_{cv} (the remaining 30%). Here, S_{cv} is called the hold-out cross validation set.

2. Train each model M_i on S_{train} only, to get some hypothesis h_i.

3. Select and output the hypothesis h_i that had the smallest error $\hat{\epsilon}_{S_{\text{cv}}}(h_i)$ on the hold out cross validation set. (Recall, $\hat{\epsilon}_{S_{\text{cv}}}(h)$ denotes the empirical error of h on the set of examples in S_{cv}.)

By testing on a set of examples S_{cv} that the models were not trained on, we obtain a better estimate of each hypothesis h_i's true generalization error, and can then pick the one with the smallest estimated generalization error. Usually, somewhere between $1/4 - 1/3$ of the data is used in the hold out cross validation set, and 30% is a typical choice.

Optionally, step 3 in the algorithm may also be replaced with selecting the model M_i according to $argmin_i \hat{\epsilon}_{S_{\text{cv}}}(h_i)$, and then retraining M_i on the entire training set S. (This is often a good idea, with one exception being learning algorithms that are be very sensitive to perturbations of the initial conditions and/or data. For these methods, M_i doing well on S_{train} does not necessarily mean it will also do well on S_{cv}, and it might be better to forgo this retraining step.)

The disadvantage of using hold out cross validation is that it "wastes" about 30% of the data. Even if we were to take the optional step of retraining the model on the entire training set, it's still as if we're trying to find a good model for a learning problem in which we had $0.7m$ training examples, rather than m training examples, since we're testing models that were trained on only $0.7m$ examples each time. While this is fine if data is abundant and/or cheap, in learning problems in which data is scarce (consider a problem with $m = 20$, say), we'd like to do something better.

Here is a method, called k-**fold cross validation**, that holds out less data each time:

1. Randomly split S into k disjoint subsets of m/k training examples each. Let's call these subsets $S_1, ..., S_k$.

2. For each model M_i, we evaluate it as follows:

   a. For $j = 1, ..., k$
      1. Train the model M_i on $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots S_k$ (i.e., train on all the data except S_j) to get some hypothesis h_{ij}.
      2. Test the hypothesis h_{ij} on S_j, to get $\hat{\epsilon}_{S_j}(h_{ij})$.

   b. The estimated generalization error of model M_i is then calculated as the average of the $\hat{\epsilon}_{S_j}(h_{ij})$'s (averaged over j).

3. Pick the model M_i with the lowest estimated generalization error, and retrain that model on the entire training set S. The resulting hypothesis is then output as our final answer.

A typical choice for the number of folds to use here would be $k = 10$. While the fraction of data held out each time is now $1/k$—much smaller than before—this procedure may also be more computationally expensive than hold-out cross validation, since we now need train to each model k times.

While $k = 10$ is a commonly used choice, in problems in which data is really scarce, sometimes we will use the extreme choice of $k = m$ in order to leave out as little data as possible each time. In this setting, we would repeatedly train on all but one of the training examples in S, and test on that held-out example. The resulting $m = k$ errors are then averaged together to obtain our estimate of the generalization error of a model. This method has its own name; since we're holding out one training example at a time, this method is called **leave-one-out cross validation.**

Finally, even though we have described the different versions of cross validation as methods for selecting a model, they can also be used more simply to evaluate a *single* model or algorithm. For example, if you have

implemented some learning algorithm and want to estimate how well it performs for your application (or if you have invented a novel learning algorithm and want to report in a technical paper how well it performs on various test sets), cross validation would give a reasonable way of doing so.

2.5.3 Feature Selection

One special and important case of model selection is called feature selection. To motivate this, imagine that you have a supervised learning problem where the number of features n is very large (perhaps $n \gg m$), but you suspect that there is only a small number of features that are "relevant" to the learning task. Even if you use a simple linear classifier (such as the perceptron) over the n input features, the VC dimension of your hypothesis class would still be $O(n)$, and thus overfitting would be a potential problem unless the training set is fairly large.

In such a setting, you can apply a feature selection algorithm to reduce the number of features. Given n features, there are 2^n possible feature subsets (since each of the n features can either be included or excluded from the subset), and thus feature selection can be posed as a model selection problem over 2^n possible models. For large values of n, it's usually too expensive to explicitly enumerate over and compare all 2^n models, and so typically some heuristic search procedure is used to find a good feature subset. The following search procedure is called **forward search**:

1. Initialize $\mathcal{F} = \varnothing$.
2. Repeat {
    a. For $i = 1, ..., n$ if $i \notin \mathcal{F}$, let $\mathcal{F}_i = \mathcal{F} \cup \{i\}$, and use some version of cross validation to evaluate features $\mathcal{F}_i$. (i.e., train your learning algorithm using only the features in $\mathcal{F}_i$, and estimate its generalization error.)
    b. Set $\mathcal{F}$ to be the best feature subset found on step (a).
3. }
4. Select and output the best feature subset that was evaluated during the entire search procedure.

The outer loop of the algorithm can be terminated either when $\mathcal{F} = \{1, ..., n\}$ is the set of all features, or when $|\mathcal{F}|$ exceeds some pre-set threshold (corresponding to the maximum number of features that you want the algorithm to consider using).

This algorithm described above one instantiation of **wrapper model feature selection**, since it is a procedure that "wraps" around your learning algorithm, and repeatedly makes calls to the learning algorithm to evaluate how well it does using different feature subsets. Aside from forward search, other search procedures can also be used. For example, **backward search** starts off with $\mathcal{F} = \{1, ..., n\}$ as the set of all features, and repeatedly deletes features one at a time (evaluating single-feature deletions in a similar manner to how forward search evaluates single-feature additions) until $\mathcal{F} = \varnothing$.

Wrapper feature selection algorithms often work quite well, but can be computationally expensive given how that they need to make many calls to the learning algorithm. Indeed, complete forward search (terminating when $\mathcal{F} = \{1, ..., n\}$) would take about $O(n^2)$ calls to the learning algorithm.

Filter feature selection methods give heuristic, but computationally much cheaper, ways of choosing a feature subset. The idea here is to compute some simple score $S(i)$ that measures how informative each feature x_i is about the class labels y. Then, we simply pick the k features with the largest scores $S(i)$.

One possible choice of the score would be define $S(i)$ to be (the absolute value of) the correlation between x_i and y, as measured on the training data. This would result in our choosing the features that are the most strongly correlated with the class labels. In practice, it is more common (particularly for discrete-valued features x_i) to choose $S(i)$ to be the **mutual information** $\mathrm{MI}(x_i, y)$ between x_i and y:

$$\mathrm{MI}(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i) p(y)}. \tag{2.162}$$

(The equation above assumes that x_i and y are binary-valued; more generally the summations would be over the domains of the variables.) The probabilities above $p(x_i, y)$, $p(x_i)$ and $p(y)$ can all be estimated according to their empirical distributions on the training set.

To gain intuition about what this score does, note that the mutual information can also be expressed as a Kullback-Leibler (KL) divergence:

$$\mathrm{MI}(x_i, y) = \mathrm{KL}(p(x_i, y) \,||\, p(x_i)\,p(y)) \tag{2.163}$$

You'll get to play more with KL-divergence in Problem set #3, but informally, this gives a measure of how different the probability distributions $p(x_i, y)$ and $p(x_i)\,p(y)$ are. If x_i and y are independent random variables, then we would have $p(x_i, y) = p(x_i)\,p(y)$, and the KL-divergence between the two distributions will be zero. This is consistent with the idea if x_i and y are independent, then x_i is clearly very "non-informative" about y, and thus the score $S(i)$ should be small. Conversely, if x_i is very "informative" about y, then their mutual information $\mathrm{MI}(x_i, y)$ would be large.

One final detail: Now that you've ranked the features according to their scores $S(i)$, how do you decide how many features k to choose? Well, one standard way to do so is to use cross validation to select among the possible values of k. For example, when applying naive Bayes to text classification—a problem where n, the vocabulary size, is usually very large—using this method to select a feature subset often results in increased classifier accuracy.

2.5.4 Bayesian statistics and regularization

In this section, we will talk about one more tool in our arsenal for our battle against overfitting.

At the beginning of the quarter, we talked about parameter fitting using maximum likelihood (ML), and chose our parameters according to

$$\theta_{\mathrm{ML}} = \underset{\theta}{argmax} \prod_{i=1}^{m} p\left(y^{(i)}|x^{(i)};\theta\right). \tag{2.164}$$

Throughout our subsequent discussions, we viewed θ as an unknown parameter of the world. This view of the θ as being *constant-valued but unknown* is taken in **frequentist** statistics. In the frequentist this view of the world, θ is not random—it just happens to be unknown—and it's our job to come up with statistical procedures (such as maximum likelihood) to try to estimate this parameter.

An alternative way to approach our parameter estimation problems is to take the **Bayesian** view of the world, and think of θ as being a *random variable* whose value is unknown. In this approach, we would specify a **prior distribution** $p(\theta)$ on θ that expresses our "prior beliefs" about the parameters. Given a training set $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, when we are asked to make a prediction on a new value of x, we can then compute the posterior distribution on the parameters

$$
\begin{aligned}
p(\theta|S) &= \frac{p(S|\theta)p(\theta)}{p(S)} \\
&= \frac{\left(\prod_{i=1}^{m} p\left(y^{(i)}|x^{(i)}, \theta\right)\right)p(\theta)}{\int_{\theta}\left(\prod_{i=1}^{m} p\left(y^{(i)}|x^{(i)}, \theta\right)p(\theta)\right)d\theta}
\end{aligned}
\tag{2.165}
$$

In the equation above, $p\left(y^{(i)}|x^{(i)}, \theta\right)$ comes from whatever model you're using for your learning problem. For example, if you are using Bayesian logistic regression, then you might choose $p\left(y^{(i)}|x^{(i)}, \theta\right) = h_\theta\left(x^{(i)}\right)^{y^{(i)}}\left(1 - h_\theta\left(x^{(i)}\right)\right)^{\left(1-y^{(i)}\right)}$, where $h_\theta\left(x^{(i)}\right) = 1/\left(1 + exp\left(-\theta^T x^{(i)}\right)\right)$.[24]

When we are given a new test example x and asked to make it prediction on it, we can compute our posterior distribution on the class label using the posterior distribution on θ:

$$p(y|x, S) = \int_{\theta} p(y|x, \theta)\,p(\theta|S)\,d\theta \tag{2.166}$$

---

[24]Since we are now viewing θ as a random variable, it is okay to condition on it value, and write "$p(y|x, \theta)$" instead of "$p(y|x;\theta)$."

In the equation above, $p\left(\theta|S\right)$ comes from (2.165). Thus, for example, if the goal is to the predict the expected value of y given x, then we would output[25]

$$E\left[y|x,S\right] = \int_y yp\left(y|x,S\right)dy \tag{2.167}$$

The procedure that we've outlined here can be thought of as doing "fully Bayesian" prediction, where our prediction is computed by taking an average with respect to the posterior $p\left(\theta|S\right)$ over θ. Unfortunately, in general it is computationally very difficult to compute this posterior distribution. This is because it requires taking integrals over the (usually high-dimensional) θ as in (2.165), and this typically cannot be done in closed-form.

Thus, in practice we will instead approximate the posterior distribution for θ. One common approximation is to replace our posterior distribution for θ (as in (2.166)) with a single point estimate. The **MAP (maximum a posteriori)** estimate for θ is given by

$$\theta_{\text{MAP}} = \underset{\theta}{argmax} \prod_{i=1}^{m} p\left(y^{(i)}|x^{(i)},\theta\right) p\left(\theta\right). \tag{2.168}$$

Note that this is the same formulas as for the ML (maximum likelihood) estimate for θ, except for the prior $p\left(\theta\right)$ term at the end.

In practical applications, a common choice for the prior $p\left(\theta\right)$ is to assume that $\theta \sim \mathcal{N}\left(0,\tau^2 I\right)$. Using this choice of prior, the fitted parameters θ_{MAP} will have smaller norm than that selected by maximum likelihood. (See Problem Set #3.) In practice, this causes the Bayesian MAP estimate to be less susceptible to overfitting than the ML estimate of the parameters. For example, Bayesian logistic regression turns out to be an effective algorithm for text classification, even though in text classification we usually have $n \gg m$.

2.6 Machine Learning Lecture 6 Course Notes[26]

2.6.1 The perceptron and large margin classifiers

In this final set of notes on learning theory, we will introduce a different model of machine learning. Specifically, we have so far been considering **batch learning** settings in which we are first given a training set to learn with, and our hypothesis h is then evaluated on separate test data. In this set of notes, we will consider the **online learning** setting in which the algorithm has to make predictions continuously even while it's learning.

In this setting, the learning algorithm is given a sequence of examples $\left(x^{(1)},y^{(1)}\right),\left(x^{(2)},y^{(2)}\right),...\left(x^{(m)},y^{(m)}\right)$ in order. Specifically, the algorithm first sees $x^{(1)}$ and is asked to predict what it thinks $y^{(1)}$ is. After making its prediction, the true value of $y^{(1)}$ is revealed to the algorithm (and the algorithm may use this information to perform some learning). The algorithm is then shown $x^{(2)}$ and again asked to make a prediction, after which $y^{(2)}$ is revealed, and it may again perform some more learning. This proceeds until we reach $\left(x^{(m)},y^{(m)}\right)$. In the online learning setting, we are interested in the total number of errors made by the algorithm during this process. Thus, it models applications in which the algorithm has to make predictions even while it's still learning.

We will give a bound on the online learning error of the perceptron algorithm. To make our subsequent derivations easier, we will use the notational convention of denoting the class labels by $y =\in \{-1,1\}$.

Recall that the perceptron algorithm has parameters $\theta \in \mathbb{R}^{n+1}$, and makes its predictions according to

$$h_\theta\left(x\right) = g\left(\theta^T x\right) \tag{2.169}$$

---

[25]The integral below would be replaced by a summation if y is discrete-valued.
[26]This content is available online at <http://cnx.org/content/m45881/1.3/>.

where

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0. \end{cases} \qquad (2.170)$$

Also, given a training example (x, y), the perceptron learning rule updates the parameters as follows. If $h_\theta(x) = y$, then it makes no change to the parameters. Otherwise, it performs the update[27]

$$\theta := \theta + yx. \qquad (2.171)$$

The following theorem gives a bound on the online learning error of the perceptron algorithm, when it is run as an online algorithm that performs an update each time it gets an example wrong. Note that the bound below on the number of errors does not have an explicit dependence on the number of examples m in the sequence, or on the dimension n of the inputs (!).

Theorem (Block, 1962, and Novikoff, 1962). Let a sequence of examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)})$ be given. Suppose that $||x^{(i)}|| \leq D$ for all i, and further that there exists a unit-length vector u ($||u||_2 = 1$) such that $y^{(i)} \cdot (u^T x^{(i)}) \geq \gamma$ for all examples in the sequence (i.e., $u^T x^{(i)} \geq \gamma$ if $y^{(i)} = 1$, and $u^T x^{(i)} \leq -\gamma$ if $y^{(i)} = -1$, so that u separates the data with a margin of at least γ). Then the total number of mistakes that the perceptron algorithm makes on this sequence is at most $(D/\gamma)^2$.

Proof. The perceptron updates its weights only on those examples on which it makes a mistake. Let $\theta^{(k)}$ be the weights that were being used when it made its k-th mistake. So, $\theta^{(1)} = \vec{0}$ (since the weights are initialized to zero), and if the k-th mistake was on the example $(x^{(i)}, y^{(i)})$, then $g\left((x^{(i)})^T \theta^{(k)}\right) \neq y^{(i)}$, which implies that

$$\left(x^{(i)}\right)^T \theta^{(k)} y^{(i)} \leq 0. \qquad (2.172)$$

Also, from the perceptron learning rule, we would have that $\theta^{(k+1)} = \theta^{(k)} + y^{(i)} x^{(i)}$.

We then have

$$\begin{aligned} \left(\theta^{(k+1)}\right)^T u &= \left(\theta^{(k)}\right)^T u + y^{(i)} \left(x^{(i)}\right)^T u \\ &\geq \left(\theta^{(k)}\right)^T u + \gamma \end{aligned} \qquad (2.173)$$

By a straightforward inductive argument, implies that

$$\left(\theta^{(k+1)}\right)^T u \geq k\gamma. \qquad (2.174)$$

Also, we have that

$$\begin{aligned} ||\theta^{(k+1)}||^2 &= ||\theta^{(k)} + y^{(i)} x^{(i)}||^2 \\ &= ||\theta^{(k)}||^2 + ||x^{(i)}||^2 + 2y^{(i)} \left(x^{(i)}\right)^T \theta^{(i)} \\ &\leq ||\theta^{(k)}||^2 + ||x^{(i)}||^2 \\ &\leq ||\theta^{(k)}||^2 + D^2 \end{aligned} \qquad (2.175)$$

The third step above used Equation (2.172). Moreover, again by applying a straightfoward inductive argument, we see that (2.175) implies

$$||\theta^{(k+1)}||^2 \leq kD^2. \qquad (2.176)$$

[27]This looks slightly different from the update rule we had written down earlier in the quarter because here we have changed the labels to be $y \in \{-1, 1\}$. Also, the learning rate parameter α was dropped. The only effect of the learning rate is to scale all the parameters θ by some fixed constant, which does not affect the behavior of the perceptron.

Putting together (2.174) and (2.175) we find that

$$
\begin{aligned}
\sqrt{k}D &\geq & ||\theta^{(k+1)}|| \\
&\geq & \left(\theta^{(k+1)}\right)^T u \\
&\geq & k\gamma.
\end{aligned}
\tag{2.177}
$$

The second inequality above follows from the fact that u is a unit-length vector (and $z^T u = ||z|| \cdot ||u|| cos\phi \leq ||z|| \cdot ||u||$, where ϕ is the angle between z and u). Our result implies that $k \leq (D/\gamma)^2$. Hence, if the perceptron made a k-th mistake, then $k \leq (D/\gamma)^2$. $\square$

2.7 Machine Learning Lecture 7a Course Notes[28]

2.7.1 The k-means clustering algorithm

In the clustering problem, we are given a training set $\{x^{(1)}, ..., x^{(m)}\}$, 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:

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

    · For every i, set

$$
c^{(i)} := \underset{j}{argmin}||x^{(i)} - \mu_j||^2.
\tag{2.178}
$$

    · For each j, set

$$
\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\}x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.
\tag{2.179}
$$

    · }

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 2.41 shows an illustration of running k-means.

---

[28]This content is available online at <http://cnx.org/content/m45898/1.3/>.
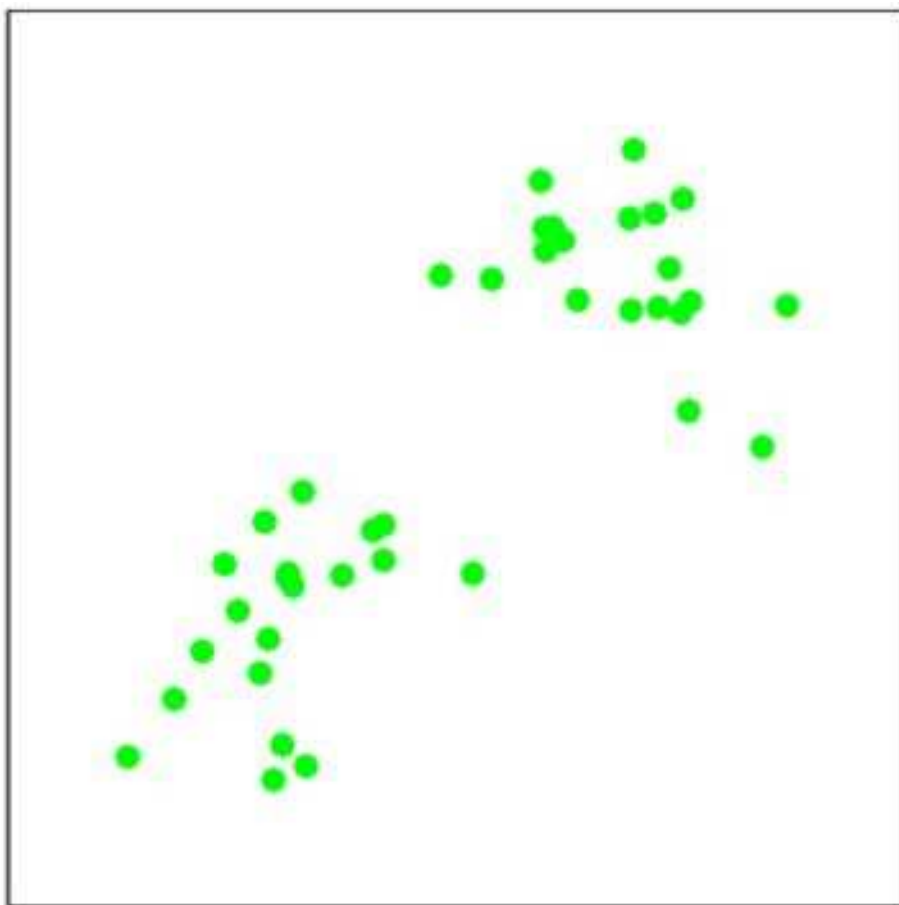
(a)
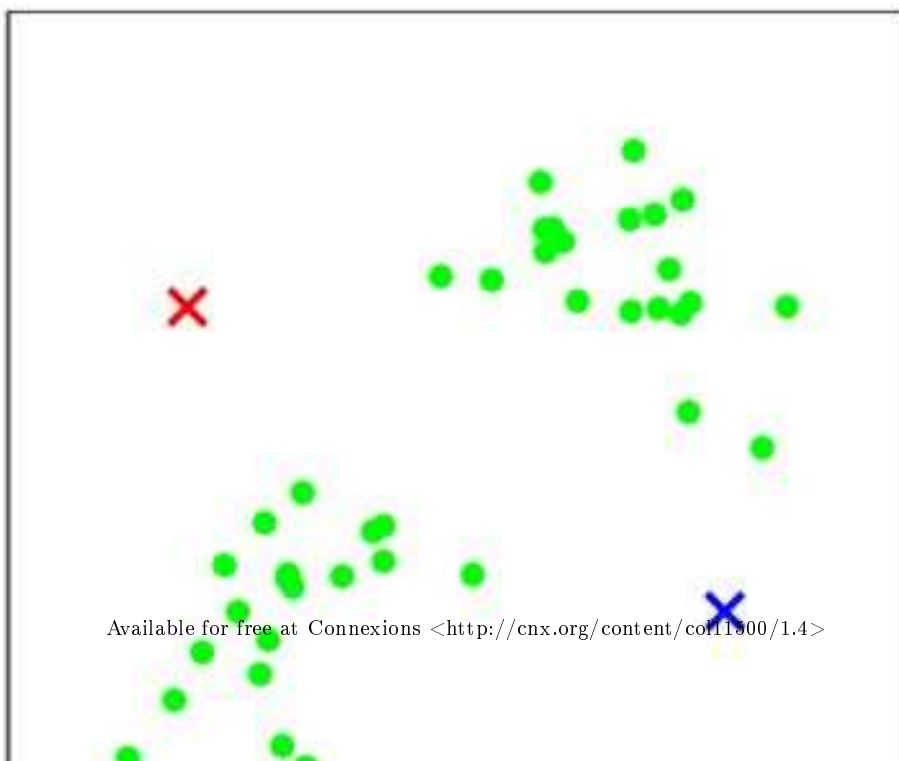
Is the k-means algorithm guaranteed to converge? Yes it is, in a certain sense. In particular, let us define the **distortion function** to be:

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2 \tag{2.180}$$

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 $J(c, \mu)$.

2.8 Machine Learning Lecture 7b Course Notes[29]

2.8.1 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 $\{x^{(1)}, ..., x^{(m)}\}$ 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 $p\left(x^{(i)}, z^{(i)}\right) = p\left(x^{(i)}|z^{(i)}\right) p\left(z^{(i)}\right)$. 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 $p\left(z^{(i)} = j\right)$), 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 $\{1, ..., k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depending 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, μ and Σ. To estimate them, we can write down the likelihood of our data:

$$
\begin{aligned}
\ell(\Phi, \mu, \Sigma) &= \sum_{i=1}^{m} \log p\left(x^{(i)}; \Phi, \mu, \Sigma\right) \\
&= \sum_{i=1}^{m} \log \sum_{z^{(i)}=1}^{k} p\left(x^{(i)}|z^{(i)}; \mu, \Sigma\right) p\left(z^{(i)}; \Phi\right).
\end{aligned}
\tag{2.181}
$$

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

$$\ell(\Phi, \mu, \Sigma) = \sum_{i=1}^{m} \log p\left(x^{(i)}|z^{(i)}; \mu, \Sigma\right) + \log p\left(z^{(i)}; \Phi\right). \tag{2.182}$$

---

[29]This content is available online at $<$http://cnx.org/content/m45893/1.3/$>$.

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

$$
\begin{aligned}
\Phi_j &= \frac{1}{m}\sum_{i=1}^m 1\{z^{(i)}=j\}, \\
\mu_j &= \frac{\sum_{i=1}^m 1\{z^{(i)}=j\}x^{(i)}}{\sum_{i=1}^m 1\{z^{(i)}=j\}}, \\
\Sigma_j &= \frac{\sum_{i=1}^m 1\{z^{(i)}=j\}\left(x^{(i)}-\mu_j\right)\left(x^{(i)}-\mu_j\right)^T}{\sum_{i=1}^m 1\{z^{(i)}=j\}}.
\end{aligned}
\tag{2.183}
$$

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.[30]

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
    $$
    w_j^{(i)} := p\left(z^{(i)}=j|x^{(i)};\Phi,\mu,\Sigma\right)
    \tag{2.184}
    $$

    · (M-step) Update the parameters:

    $$
    \begin{aligned}
    \Phi_j &:= \frac{1}{m}\sum_{i=1}^m w_j^{(i)}, \\
    \mu_j &:= \frac{\sum_{i=1}^m w_j^{(i)}x^{(i)}}{\sum_{i=1}^m w_j^{(i)}}, \\
    \Sigma_j &:= \frac{\sum_{i=1}^m w_j^{(i)}\left(x^{(i)}-\mu_j\right)\left(x^{(i)}-\mu_j\right)^T}{\sum_{i=1}^m w_j^{(i)}}
    \end{aligned}
    \tag{2.185}
    $$

- }

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:

$$
p\left(z^{(i)}=j|x^{(i)};\Phi,\mu,\Sigma\right) = \frac{p\left(x^{(i)}|z^{(i)}=j;\mu,\Sigma\right)p\left(z^{(i)}=j;\Phi\right)}{\sum_{l=1}^k p\left(x^{(i)}|z^{(i)}=l;\mu,\Sigma\right)p\left(z^{(i)}=l;\Phi\right)}
\tag{2.186}
$$

Here, $p\left(x^{(i)}|z^{(i)}=j;\mu,\Sigma\right)$ is given by evaluating the density of a Gaussian with mean μ_j and covariance Σ_j at $x^{(i)}$; $p\left(z^{(i)}=j;\Phi\right)$ is given by Φ_j, and so on. The values $w_j^{(i)}$ calculated in the E-step represent our "soft" guesses[31] 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.

---

[30] 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.

[31] 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 $\{1, ..., k\}$).

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 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.

2.9 Machine Learning Lecture 8 Course Notes[32]

2.9.1 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**

2.9.2 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 positive 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:

$$E[f(X)] \geq f(EX). \tag{2.187}$$

Moreover, if f is strictly convex, then $E[f(X)] = f(EX)$ holds true if and only if $X = E[X]$ 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, $f(EX) = f(E[X])$.

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

---

[32] This content is available online at <http://cnx.org/content/m45901/1.3/>.
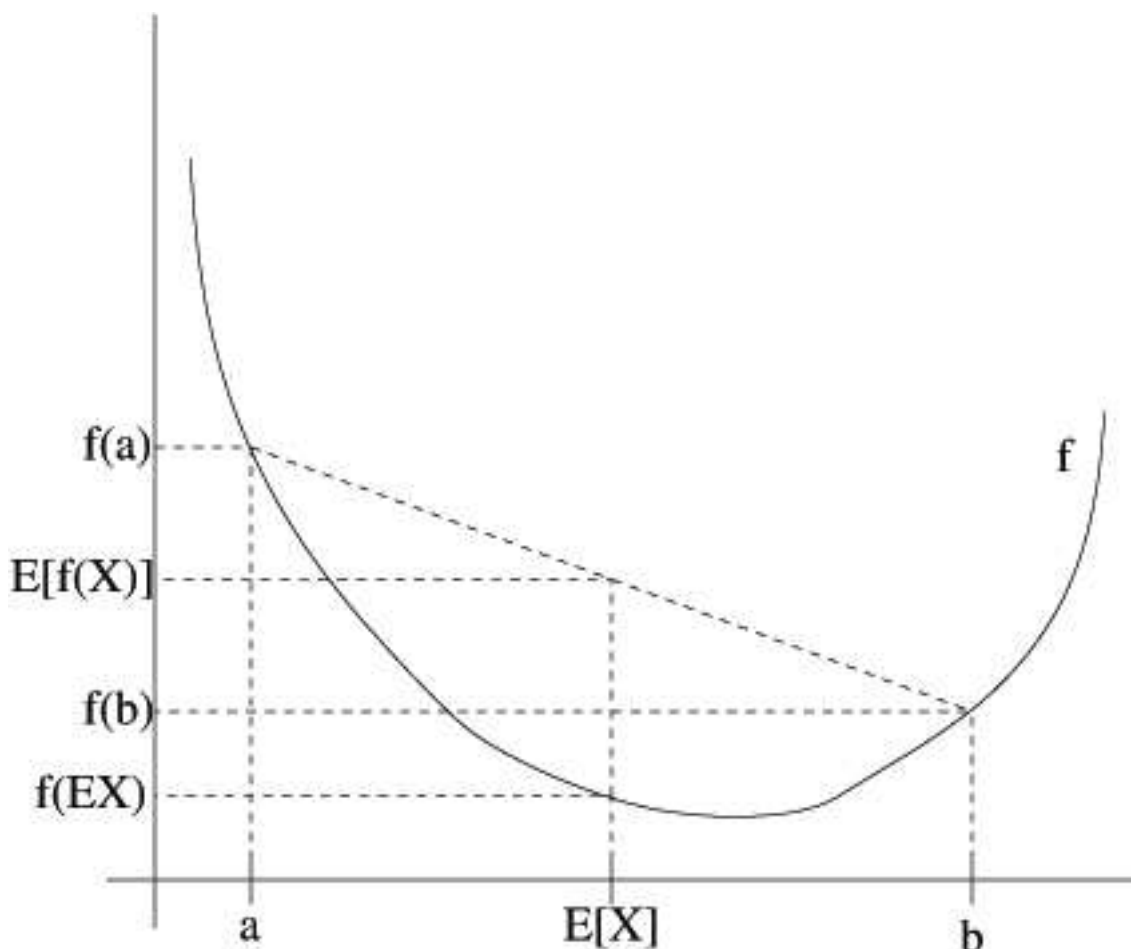
Figure 2.42

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 $f(E[X])$ indicated on the y-axis. Moreover, the value $E[f(X)]$ 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 $E[f(X)] \geq f(EX)$.

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 ($E[f(X)] \leq f(EX)$, etc.).

2.9.3 The EM algorithm

Suppose we have an estimation problem in which we have a training set $\{x^{(1)}, ..., x^{(m)}\}$ 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

$$
\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{m} \log p(x; \theta) \\
&= \sum_{i=1}^{m} \log \sum_{z} p(x, z; \theta).
\end{aligned}
\tag{2.188}
$$

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:[33]

$$
\sum_i \log p\left(x^{(i)}; \theta\right) = \sum_i \log \sum_{z^{(i)}} p\left(x^{(i)}, z^{(i)}; \theta\right)
\tag{2.189}
$$

$$
= \sum_i \log \sum_{z^{(i)}} Q_i\left(z^{(i)}\right) \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)}
\tag{2.190}
$$

$$
\geq \sum_i \sum_{z^{(i)}} Q_i\left(z^{(i)}\right) \log \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)}
\tag{2.191}
$$

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

$$
\sum_{z^{(i)}} Q_i\left(z^{(i)}\right) \left[\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)} \right]
\tag{2.192}
$$

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

$$
f\left(E_{z^{(i)} \sim Q_i} \left[\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)} \right] \right) \geq E_{z^{(i)} \sim Q_i} \left[f\left(\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)} \right) \right],
\tag{2.193}
$$

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 (2.189) to (2.191).

Now, for *any* set of distributions Q_i, the formula (2.191) 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 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

$$
\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)} = c
\tag{2.194}
$$

---

[33] 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.

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

$$Q_i\left(z^{(i)}\right) \propto p\left(x^{(i)}, z^{(i)}; \theta\right). \tag{2.195}$$

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

$$
\begin{aligned}
Q_i\left(z^{(i)}\right) &= \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{\sum_z p\left(x^{(i)}, z; \theta\right)} \\
&= \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{p\left(x^{(i)}; \theta\right)} \\
&= p\left(z^{(i)}|x^{(i)}; \theta\right)
\end{aligned}
\tag{2.196}
$$

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 (2.191) 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 (2.191) 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

$$Q_i\left(z^{(i)}\right) := p\left(z^{(i)}|x^{(i)}; \theta\right). \tag{2.197}$$

  - (M-step) Set

$$\theta := \underset{\theta}{argmax} \sum_i \sum_{z^{(i)}} Q_i\left(z^{(i)}\right) log\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)}. \tag{2.198}$$

- }

How will 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 $\ell\left(\theta^{(t)}\right) \leq \ell\left(\theta^{(t+1)}\right)$, 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 $Q_i^{(t)}\left(z^{(i)}\right) := p\left(z^{(i)}|x^{(i)}; \theta^{(t)}\right)$. We saw earlier that this choice ensures that Jensen's inequality, as applied to get (2.191), holds with equality, and hence

$$\ell\left(\theta^{(t)}\right) = \sum_i \sum_{z^{(i)}} Q_i^{(t)}\left(z^{(i)}\right) log\frac{p\left(x^{(i)}, z^{(i)}; \theta^{(t)}\right)}{Q_i^{(t)}\left(z^{(i)}\right)}. \tag{2.199}$$

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

$$
\begin{aligned}
\ell\left(\theta^{(t+1)}\right) &\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}\left(z^{(i)}\right) log\frac{p\left(x^{(i)}, z^{(i)}; \theta^{(t+1)}\right)}{Q_i^{(t)}\left(z^{(i)}\right)} \tag{2.200} \\
&\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}\left(z^{(i)}\right) log\frac{p\left(x^{(i)}, z^{(i)}; \theta^{(t)}\right)}{Q_i^{(t)}\left(z^{(i)}\right)} \tag{2.201} \\
&= \ell\left(\theta^{(t)}\right) \tag{2.202}
\end{aligned}
$$

This first inequality comes from the fact that

$$\ell\left(\theta\right) \geq \sum_i \sum_{z^{(i)}} Q_i\left(z^{(i)}\right) log\frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i\left(z^{(i)}\right)} \tag{2.203}$$

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 (2.200), we used the fact that $\theta^{(t+1)}$ is chosen explicitly to be

$$\underset{\theta}{argmax} \sum_i \sum_{z^{(i)}} Q_i \left(z^{(i)} \right) log \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i \left(z^{(i)} \right)}, \tag{2.204}$$

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 (2.202) 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

$$J(Q, \theta) = \sum_i \sum_{z^{(i)}} Q_i \left(z^{(i)} \right) log \frac{p\left(x^{(i)}, z^{(i)}; \theta\right)}{Q_i \left(z^{(i)} \right)}, \tag{2.205}$$

then 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.

2.9.4 Mixture of Gaussians revisited

Armed with our general definition of the EM algorithm, let's 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

$$w_j^{(i)} = Q_i \left(z^{(i)} = j \right) = P \left(z^{(i)} = j | x^{(i)}; \Phi, \mu, \Sigma \right). \tag{2.206}$$

Here, "$Q_i \left(z^{(i)} = j \right)$" 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

$$\begin{aligned}
& \sum_{i=1}^m \sum_{z^{(i)}} Q_i \left(z^{(i)} \right) log \frac{p\left(x^{(i)}, z^{(i)}; \Phi, \mu, \Sigma\right)}{Q_i\left(z^{(i)}\right)} \\
& = \sum_{i=1}^m \sum_{j=1}^k Q_i \left(z^{(i)} = j \right) log \frac{p\left(x^{(i)} | z^{(i)}=j; \mu, \Sigma\right) p\left(z^{(i)}=j; \Phi\right)}{Q_i\left(z^{(i)}=j\right)} \\
& = \sum_{i=1}^m \sum_{j=1}^k w_j^{(i)} log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} exp\left(-\frac{1}{2}\left(x^{(i)}-\mu_j\right)^T \Sigma_j^{-1}\left(x^{(i)}-\mu_j\right)\right) \cdot \Phi_j}{w_j^{(i)}}
\end{aligned} \tag{2.207}$$

Let's maximize this with respect to μ_l. If we take the derivative with respect to μ_l, we find

$$\begin{aligned}
& \nabla_{\mu_l} \sum_{i=1}^m \sum_{j=1}^k w_j^{(i)} log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} exp\left(-\frac{1}{2}\left(x^{(i)}-\mu_j\right)^T \Sigma_j^{-1}\left(x^{(i)}-\mu_j\right)\right) \cdot \Phi_j}{w_j^{(i)}} \\
= & -\nabla_{\mu_l} \sum_{i=1}^m \sum_{j=1}^k w_j^{(i)} \frac{1}{2}\left(x^{(i)}-\mu_j\right)^T \Sigma_j^{-1}\left(x^{(i)}-\mu_j\right) \\
= & \frac{1}{2} \sum_{i=1}^m w_l^{(i)} \nabla_{\mu_l} 2\mu_l^T \Sigma_l^{-1} x^{(i)} - \mu_l^T \Sigma_l^{-1} \mu_l \\
= & \sum_{i=1}^m w_l^{(i)} \left(\Sigma_l^{-1} x^{(i)} - \Sigma_l^{-1} \mu_l \right)
\end{aligned} \tag{2.208}$$

Setting this to zero and solving for μ_l therefore yields the update rule

$$\mu_l := \frac{\sum_{i=1}^m w_l^{(i)} x^{(i)}}{\sum_{i=1}^m w_l^{(i)}}, \tag{2.209}$$

which was what we had in the previous set of notes.

Let's 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

$$\sum_{i=1}^{m}\sum_{j=1}^{k} w_j^{(i)} log \Phi_j. \tag{2.210}$$

However, there is an additional constraint that the Φ_j's sum to 1, since they represent the probabilities $\Phi_j = p\left(z^{(i)} = j; \Phi\right)$. To deal with the constraint that $\sum_{j=1}^{k} \Phi_j = 1$, we construct the Lagrangian

$$\mathcal{L}\left(\Phi\right) = \sum_{i=1}^{m}\sum_{j=1}^{k} w_j^{(i)} log \Phi_j + \beta \left(\sum_{j=1}^{k} \Phi_j - 1\right), \tag{2.211}$$

where β is the Lagrange multiplier.[34] Taking derivatives, we find

$$\frac{\partial}{\partial \Phi_j} \mathcal{L}\left(\Phi\right) = \sum_{i=1}^{m} \frac{w_j^{(i)}}{\Phi_j} + 1 \tag{2.212}$$

Setting this to zero and solving, we get

$$\Phi_j = \frac{\sum_{i=1}^{m} w_j^{(i)}}{-\beta} \tag{2.213}$$

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 $w_j^{(i)} = Q_i\left(z^{(i)} = j\right)$, and since probabilities sum to 1, $\sum_j w_j^{(i)} = 1$.) We therefore have our M-step updates for the parameters Φ_j:

$$\Phi_j := \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)}. \tag{2.214}$$

The derivation for the M-step updates to Σ_j are also entirely straightforward.

2.10 Machine Learning Lecture 9 Course Notes[35]

2.10.1 Factor analysis

When we have data $x^{(i)} \in \mathbb{R}^n$ that comes from a mixture of several Gaussians, the EM algorithm can be applied to fit a mixture model. In this setting, we usually imagine problems where we have sufficient data to be able to discern the multiple-Gaussian structure in the data. For instance, this would be the case if our training set size m was significantly larger than the dimension n of the data.

Now, consider a setting in which $n \gg m$. In such a problem, it might be difficult to model the data even with a single Gaussian, much less a mixture of Gaussian. Specifically, since the m data points span only a low-dimensional subspace of $\mathbb{R}^n$, if we model the data as Gaussian, and estimate the mean and covariance using the usual maximum likelihood estimators,

$$\begin{array}{rcl} \mu & = & \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \Sigma & = & \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - \mu\right)\left(x^{(i)} - \mu\right)^T, \end{array} \tag{2.215}$$

---

[34]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.

[35]This content is available online at $<$http://cnx.org/content/m45915/1.3/$>$.

we would find that the matrix Σ is singular. This means that Σ^{-1} does not exist, and $1/|\Sigma|^{1/2} = 1/0$. But both of these terms are needed in computing the usual density of a multivariate Gaussian distribution. Another way of stating this difficulty is that maximum likelihood estimates of the parameters result in a Gaussian that places all of its probability in the affine space spanned by the data,[36] and this corresponds to a singular covariance matrix.

More generally, unless m exceeds n by some reasonable amount, the maximum likelihood estimates of the mean and covariance may be quite poor. Nonetheless, we would still like to be able to fit a reasonable Gaussian model to the data, and perhaps capture some interesting covariance structure in the data. How can we do this?

In the next section, we begin by reviewing two possible restrictions on Σ, ones that allow us to fit Σ with small amounts of data but neither of which will give a satisfactory solution to our problem. We next discuss some properties of Gaussians that will be needed later; specifically, how to find marginal and conditonal distributions of Gaussians. Finally, we present the factor analysis model, and EM for it.

2.10.2 Restrictions of Σ

If we do not have sufficient data to fit a full covariance matrix, we may place some restrictions on the space of matrices Σ that we will consider. For instance, we may choose to fit a covariance matrix Σ that is diagonal. In this setting, the reader may easily verify that the maximum likelihood estimate of the covariance matrix is given by the diagonal matrix Σ satisfying

$$\Sigma_{jj} = \frac{1}{m} \sum_{i=1}^{m} \left(x_j^{(i)} - \mu_j \right)^2. \tag{2.216}$$

Thus, Σ_{jj} is just the empirical estimate of the variance of the j-th coordinate of the data.

Recall that the contours of a Gaussian density are ellipses. A diagonal Σ corresponds to a Gaussian where the major axes of these ellipses are axis-aligned.

Sometimes, we may place a further restriction on the covariance matrix that not only must it be diagonal, but its diagonal entries must all be equal. In this setting, we have $\Sigma = \sigma^2 I$, where σ^2 is the parameter under our control. The maximum likelihood estimate of σ^2 can be found to be:

$$\sigma^2 = \frac{1}{mn} \sum_{j=1}^{n} \sum_{i=1}^{m} \left(x_j^{(i)} - \mu_j \right)^2. \tag{2.217}$$

This model corresponds to using Gaussians whose densities have contours that are circles (in 2 dimensions; or spheres/hyperspheres in higher dimensions).

If we were fitting a full, unconstrained, covariance matrix Σ to data, it was necessary that $m \geq n + 1$ in order for the maximum likelihood estimate of Σ not to be singular. Under either of the two restrictions above, we may obtain non-singular Σ when $m \geq 2$.

However, restricting Σ to be diagonal also means modeling the different coordinates x_i, x_j of the data as being uncorrelated and independent. Often, it would be nice to be able to capture some interesting correlation structure in the data. If we were to use either of the restrictions on Σ described above, we would therefore fail to do so. In this set of notes, we will describe the factor analysis model, which uses more parameters than the diagonal Σ and captures some correlations in the data, but also without having to fit a full covariance matrix.

2.10.3 Marginals and conditionals of Gaussians

Before describing factor analysis, we digress to talk about how to find conditional and marginal distributions of random variables with a joint multivariate Gaussian distribution.

---

[36]This is the set of points x satisfying $x = \sum_{i=1}^{m} \alpha_i x^{(i)}$, for some α_i's so that $\sum_{i=1}^{m} \alpha_1 = 1$.

Suppose we have a vector-valued random variable

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \tag{2.218}$$

where $x_1 \in \mathbb{R}^r$, $x_2 \in \mathbb{R}^s$, and $x \in \mathbb{R}^{r+s}$. Suppose $x \sim \mathcal{N}(\mu, \Sigma)$, where

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}. \tag{2.219}$$

Here, $\mu_1 \in \mathbb{R}^r$, $\mu_2 \in \mathbb{R}^s$, $\Sigma_{11} \in \mathbb{R}^{r \times r}$, $\Sigma_{12} \in \mathbb{R}^{r \times s}$, and so on. Note that since covariance matrices are symmetric, $\Sigma_{12} = \Sigma_{21}^T$.

Under our assumptions, x_1 and x_2 are jointly multivariate Gaussian. What is the marginal distribution of x_1? It is not hard to see that $E[x_1] = \mu_1$, and that $\text{Cov}(x_1) = E[(x_1 - \mu_1)(x_1 - \mu_1)_T] = \Sigma_{11}$. To see that the latter is true, note that by definition of the joint covariance of x_1 and x_2, we have that

$$
\begin{aligned}
\text{Cov}(x) &= \Sigma \\
&= \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \\
&= E\left[(x - \mu)(x - \mu)^T\right] \\
&= E\left[\begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix}^T \right] \\
&= E\left[\begin{matrix} (x_1 - \mu_1)(x_1 - \mu_1)^T & (x_1 - \mu_1)(x_2 - \mu_2)^T \\ (x_2 - \mu_2)(x_1 - \mu_1)^T & (x_2 - \mu_2)(x_2 - \mu_2)^T \end{matrix} \right].
\end{aligned} \tag{2.220}
$$

Matching the upper-left subblocks in the matrices in the second and the last lines above gives the result.

Since marginal distributions of Gaussians are themselves Gaussian, we therefore have that the marginal distribution of x_1 is given by $x_1 \sim \mathcal{N}(\mu_1, \Sigma_{11})$.

Also, we can ask, what is the conditional distribution of x_1 given x_2? By referring to the definition of the multivariate Gaussian distribution, it can be shown that $x_1|x_2 \sim \mathcal{N}(\mu_{1|2}, \Sigma_{1|2})$, where

$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2), \tag{2.221}$$

$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \tag{2.222}$$

When working with the factor analysis model in the next section, these formulas for finding conditional and marginal distributions of Gaussians will be very useful.

2.10.4 The Factor analysis model

In the factor analysis model, we posit a joint distribution on (x, z) as follows, where $z \in \mathbb{R}^k$ is a latent random variable:

$$
\begin{aligned}
z &\sim \mathcal{N}(0, I) \\
x|z &\sim \mathcal{N}(\mu + \Lambda z, \Psi).
\end{aligned} \tag{2.223}
$$

Here, the parameters of our model are the vector $\mu \in \mathbb{R}^n$, the matrix $\Lambda \in \mathbb{R}^{n \times k}$, and the diagonal matrix $\Psi \in \mathbb{R}^{n \times n}$. The value of k is usually chosen to be smaller than n.

Thus, we imagine that each datapoint $x^{(i)}$ is generated by sampling a k dimension multivariate Gaussian $z^{(i)}$. Then, it is mapped to a k-dimensional affine space of $\mathbb{R}^n$ by computing $\mu + \Lambda z^{(i)}$. Lastly, $x^{(i)}$ is generated by adding covariance Ψ noise to $\mu + \Lambda z^{(i)}$.

Equivalently (convince yourself that this is the case), we can therefore also define the factor analysis model according to

$$
\begin{aligned}
z &\sim & \mathcal{N}\left(0, I\right) \\
\varepsilon &\sim & \mathcal{N}\left(0, \Psi\right) \\
x &= & \mu + \Lambda z + \varepsilon.
\end{aligned}
\tag{2.224}
$$

where ε and z are independent.

Let's work out exactly what distribution our model defines. Our random variables z and x have a joint Gaussian distribution

$$
\left[\begin{array}{c} z \\ x \end{array}\right] \sim \mathcal{N}\left(\mu_{zx}, \Sigma\right).
\tag{2.225}
$$

We will now find μ_{zx} and Σ.

We know that $E\left[z\right] = 0$, from the fact that $z \sim \mathcal{N}\left(0, I\right)$. Also, we have that

$$
\begin{aligned}
E\left[x\right] &= & E\left[\mu + \Lambda z + \varepsilon\right] \\
&= & \mu + \Lambda E\left[z\right] + E\left[\varepsilon\right] \\
&= & \mu.
\end{aligned}
\tag{2.226}
$$

Putting these together, we obtain

$$
\mu_{zx} = \left[\begin{array}{c} \vec{0} \\ \mu \end{array}\right]
\tag{2.227}
$$

Next, to find, Σ, we need to calculate $\Sigma_{zz} = E\left[\left(z - E\left[z\right]\right)\left(z - E\left[z\right]\right)^T\right]$ (the upper-left block of Σ), $\Sigma_{zx} = E\left[\left(z - E\left[z\right]\right)\left(x - E\left[x\right]\right)^T\right]$ (upper-right block), and $\Sigma_{xx} = E\left[\left(x - E\left[x\right]\right)\left(x - E\left[x\right]\right)^T\right]$ (lower-right block).

Now, since $z \sim \mathcal{N}\left(0, I\right)$, we easily find that $\Sigma_{zz} = \text{Cov}\left(z\right) = I$. Also,

$$
\begin{aligned}
E\left[\left(z - E\left[z\right]\right)\left(x - E\left[x\right]\right)^T\right] &= & E\left[z(\mu + \Lambda z + \varepsilon - \mu)^T\right] \\
&= & E\left[zz^T\right]\Lambda^T + E\left[z\varepsilon^T\right] \\
&= & \Lambda^T.
\end{aligned}
\tag{2.228}
$$

In the last step, we used the fact that $E\left[zz^T\right] = \text{Cov}\left(z\right)$ (since z has zero mean), and $E\left[z\varepsilon^T\right] = E\left[z\right]E\left[\varepsilon^T\right] = 0$ (since z and ε are independent, and hence the expectation of their product is the product of their expectations). Similarly, we can find Σ_{xx} as follows:

$$
\begin{aligned}
E\left[\left(x - E\left[x\right]\right)\left(x - E\left[x\right]\right)^T\right] &= & E\left[\left(\mu + \Lambda z + \varepsilon - \mu\right)\left(\mu + \Lambda z + \varepsilon - \mu\right)^T\right] \\
&= & E\left[\Lambda zz^T \Lambda^T + \varepsilon z^T \Lambda^T + \Lambda z\varepsilon^T + \varepsilon\varepsilon^T\right] \\
&= & \Lambda E\left[zz^T\right]\Lambda^T + E\left[\varepsilon\varepsilon^T\right] \\
&= & \Lambda\Lambda^T + \Psi.
\end{aligned}
\tag{2.229}
$$

Putting everything together, we therefore have that

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N} \left(\begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^T \\ \Lambda & \Lambda\Lambda^T + \Psi \end{bmatrix} \right). \tag{2.230}$$

Hence, we also see that the marginal distribution of x is given by $x \sim \mathcal{N}\left(\mu, \Lambda\Lambda^T + \Psi\right)$. Thus, given a training set $\{x^{(i)}; i = 1, ..., m\}$, we can write down the log likelihood of the parameters:

$$\ell\left(\mu, \Lambda, \Psi\right) = log \prod_{i=1}^{m} \frac{1}{(2\pi)^{n/2}|\Lambda\Lambda^T + \Psi|^{1/2}} exp\left(-\frac{1}{2}\left(x^{(i)} - \mu\right)^T\left(\Lambda\Lambda^T + \Psi\right)^{-1}\left(x^{(i)} - \mu\right)\right). \tag{2.231}$$

To perform maximum likelihood estimation, we would like to maximize this quantity with respect to the parameters. But maximizing this formula explicitly is hard (try it yourself), and we are aware of no algorithm that does so in closed-form. So, we will instead use to the EM algorithm. In the next section, we derive EM for factor analysis.

2.10.5 EM for factor analysis

The derivation for the E-step is easy. We need to compute $Q_i\left(z^{(i)}\right) = p\left(z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi\right)$. By substituting the distribution given in Equation (2.230) into the formulas (2.221)-(2.222) used for finding the conditional distribution of a Gaussian, we find that $z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi \sim \mathcal{N}\left(\mu_{z^{(i)}|x^{(i)}}, \Sigma_{z^{(i)}|x^{(i)}}\right)$, where

$$\begin{aligned} \mu_{z^{(i)}|x^{(i)}} &= \Lambda^T\left(\Lambda\Lambda^T + \Psi\right)^{-1}\left(x^{(i)} - \mu\right), \\ \Sigma_{z^{(i)}|x^{(i)}} &= I - \Lambda^T\left(\Lambda\Lambda^T + \Psi\right)^{-1}\Lambda. \end{aligned} \tag{2.232}$$

So, using these definitions for $\mu_{z^{(i)}|x^{(i)}}$ and $\Sigma_{z^{(i)}|x^{(i)}}$, we have

$$Q_i\left(z^{(i)}\right) = \frac{1}{(2\pi)^{k/2}\left|\Sigma_{z^{(i)}|x^{(i)}}\right|^{1/2}} exp\left(-\frac{1}{2}\left(z^{(i)} - \mu_{z^{(i)}|x^{(i)}}\right)^T \Sigma_{z^{(i)}|x^{(i)}}^{-1}\left(z^{(i)} - \mu_{z^{(i)}|x^{(i)}}\right)\right). \tag{2.233}$$

Let's now work out the M-step. Here, we need to maximize

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i\left(z^{(i)}\right) log \frac{p\left(x^{(i)}, z^{(i)}; \mu, \Lambda, \Psi\right)}{Q_i\left(z^{(i)}\right)} dz^{(i)} \tag{2.234}$$

with respect to the parameters μ, Λ, Ψ. We will work out only the optimization with respect to Λ, and leave the derivations of the updates for μ and Ψ as an exercise to the reader.

We can simplify Equation (2.234) as follows:

$$\begin{aligned} &\sum_{i=1}^{m} \int_{z^{(i)}} Q_i\left(z^{(i)}\right) \left[log\, p\left(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi\right) + log\, p\left(z^{(i)}\right) - log\, Q_i\left(z^{(i)}\right)\right] dz^{(i)} \\ = &\sum_{i=1}^{m} E_{z^{(i)}\sim Q_i} \left[log\, p\left(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi\right) + log\, p\left(z^{(i)}\right) - log\, Q_i\left(z^{(i)}\right)\right] \end{aligned} \tag{2.235}$$

Here, the "$z^{(i)} \sim Q_i$" subscript indicates that the expectation is with respect to $z^{(i)}$ drawn from Q_i. In the subsequent development, we will omit this subscript when there is no risk of ambiguity. Dropping terms that do not depend on the parameters, we find that we need to maximize:

$$\begin{aligned} &\sum_{i=1}^{m} E\left[log\, p\left(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi\right)\right] \\ = &\sum_{i=1}^{m} E\left[log\, \frac{1}{(2\pi)^{n/2}|\Psi|^{1/2}} exp\left(-\frac{1}{2}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)^T\Psi^{-1}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)\right)\right] \\ = &\sum_{i=1}^{m} E\left[-\frac{1}{2}log|\Psi| - \frac{n}{2}log\left(2\pi\right) - \frac{1}{2}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)^T\Psi^{-1}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)\right] \end{aligned} \tag{2.236}$$

Let's maximize this with respect to Λ. Only the last term above depends on Λ. Taking derivatives, and using the facts that $\operatorname{tr} a = a$ (for $a \in \mathbb{R}$), $\operatorname{tr} AB = \operatorname{tr} BA$, and $\nabla_A \operatorname{tr} ABA^T C = CAB + C^T AB$, we get:

$$
\begin{aligned}
&\nabla_\Lambda \sum_{i=1}^m -E\left[\tfrac{1}{2}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)^T \Psi^{-1}\left(x^{(i)} - \mu - \Lambda z^{(i)}\right)\right] \\
=\ & \sum_{i=1}^m \nabla_\Lambda E\left[-\operatorname{tr}\tfrac{1}{2}z^{(i)T}\Lambda^T \Psi^{-1}\Lambda z^{(i)} + \operatorname{tr} z^{(i)T}\Lambda^T\Psi^{-1}\left(x^{(i)} - \mu\right)\right] \\
=\ & \sum_{i=1}^m \nabla_\Lambda E\left[-\operatorname{tr}\tfrac{1}{2}\Lambda^T \Psi^{-1}\Lambda z^{(i)}z^{(i)T} + \operatorname{tr}\Lambda^T\Psi^{-1}\left(x^{(i)} - \mu\right)z^{(i)T}\right] \\
=\ & \sum_{i=1}^m E\left[-\Psi^{-1}\Lambda z^{(i)}z^{(i)T} + \Psi^{-1}\left(x^{(i)} - \mu\right)z^{(i)T}\right]
\end{aligned}
\tag{2.237}
$$

Setting this to zero and simplifying, we get:

$$
\sum_{i=1}^m \Lambda E_{z^{(i)} \sim Q_i}\left[z^{(i)}z^{(i)T}\right] = \sum_{i=1}^m \left(x^{(i)} - \mu\right) E_{z^{(i)} \sim Q_i}\left[z^{(i)T}\right].
\tag{2.238}
$$

Hence, solving for Λ, we obtain

$$
\Lambda = \left(\sum_{i=1}^m \left(x^{(i)} - \mu\right) E_{z^{(i)} \sim Q_i}\left[z^{(i)T}\right]\right)\left(\sum_{i=1}^m E_{z^{(i)} \sim Q_i}\left[z^{(i)}z^{(i)T}\right]\right)^{-1}.
\tag{2.239}
$$

It is interesting to note the close relationship between this equation and the normal equation that we'd derived for least squares regression,

$$
\text{``}\theta^T = \left(y^T X\right)\left(X^T X\right)^{-1}.\text{''}
\tag{2.240}
$$

The analogy is that here, the x's are a linear function of the z's (plus noise). Given the "guesses" for z that the E-step has found, we will now try to estimate the unknown linearity Λ relating the x's and z's. It is therefore no surprise that we obtain something similar to the normal equation. There is, however, one important difference between this and an algorithm that performs least squares using just the "best guesses" of the z's; we will see this difference shortly.

To complete our M-step update, let's work out the values of the expectations in Equation (2.239). From our definition of Q_i being Gaussian with mean $\mu_{z^{(i)}|x^{(i)}}$ and covariance $\Sigma_{z^{(i)}|x^{(i)}}$, we easily find

$$
\begin{aligned}
E_{z^{(i)} \sim Q_i}\left[z^{(i)T}\right] &= \mu_{z^{(i)}|x^{(i)}}^T \\
E_{z^{(i)} \sim Q_i}\left[z^{(i)}z^{(i)T}\right] &= \mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}}.
\end{aligned}
\tag{2.241}
$$

The latter comes from the fact that, for a random variable Y, $\operatorname{Cov}(Y) = E\left[YY^T\right] - E\left[Y\right]E[Y]^T$, and hence $E\left[YY^T\right] = E\left[Y\right]E[Y]^T + \operatorname{Cov}(Y)$. Substituting this back into Equation (2.239), we get the M-step update for Λ:

$$
\Lambda = \left(\sum_{i=1}^m \left(x^{(i)} - \mu\right)\mu_{z^{(i)}|x^{(i)}}^T\right)\left(\sum_{i=1}^m \mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}}\right)^{-1}.
\tag{2.242}
$$

It is important to note the presence of the $\Sigma_{z^{(i)}|x^{(i)}}$ on the right hand side of this equation. This is the covariance in the posterior distribution $p\left(z^{(i)}|x^{(i)}\right)$ of $z^{(i)}$ give $x^{(i)}$, and the M-step must take into account this uncertainty about $z^{(i)}$ in the posterior. A common mistake in deriving EM is to assume that in the E-step, we need to calculate only expectation $E\left[z\right]$ of the latent random variable z, and then plug that into the optimization in the M-step everywhere z occurs. While this worked for simple problems such as the mixture of Gaussians, in our derivation for factor analysis, we needed $E\left[zz^T\right]$ as well $E\left[z\right]$; and as we saw, $E\left[zz^T\right]$ and $E\left[z\right]E[z]^T$ differ by the quantity $\Sigma_{z|x}$. Thus, the M-step update must take into account the covariance of z in the posterior distribution $p\left(z^{(i)}|x^{(i)}\right)$.

Lastly, we can also find the M-step optimizations for the parameters μ and Ψ. It is not hard to show that the first is given by

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}. \tag{2.243}$$

Since this doesn't change as the parameters are varied (i.e., unlike the update for Λ, the right hand side does not depend on $Q_i\left(z^{(i)}\right) = p\left(z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi\right)$, which in turn depends on the parameters), this can be calculated just once and needs not be further updated as the algorithm is run. Similarly, the diagonal Ψ can be found by calculating

$$\Phi = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} x^{(i)T} - x^{(i)} \mu_{z^{(i)}|x^{(i)}}^{T} \Lambda^T - \Lambda \mu_{z^{(i)}|x^{(i)}} x^{(i)T} + \Lambda \left(\mu_{z^{(i)}|x^{(i)}} \mu_{z^{(i)}|x^{(i)}}^{T} + \Sigma_{z^{(i)}|x^{(i)}} \right) \Lambda^T, \tag{2.244}$$

and setting $\Psi_{ii} = \Phi_{ii}$ (i.e., letting Ψ be the diagonal matrix containing only the diagonal entries of Φ).

2.11 Machine Learning Lecture 10 Course Notes[37]

2.11.1 Principal components analysis

In our discussion of factor analysis, we gave a way to model data $x \in \mathbb{R}^n$ as "approximately" lying in some k-dimension subspace, where $k \ll n$. Specifically, we imagined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$, and then adding Ψ-covariance noise. Factor analysis is based on a probabilistic model, and parameter estimation used the iterative EM algorithm.

In this set of notes, we will develop a method, Principal Components Analysis (PCA), that also tries to identify the subspace in which the data approximately lies. However, PCA will do so more directly, and will require only an eigenvector calculation (easily done with the `eig` function in Matlab), and does not need to resort to EM.

Suppose we are given a dataset $\{x^{(i)}; i = 1, ..., m\}$ of attributes of m different types of automobiles, such as their maximum speed, turn radius, and so on. Let $x^{(i)} \in \mathbb{R}^n$ for each i $(n \ll m)$. But unknown to us, two different attributes—some x_i and x_j—respectively give a car's maximum speed measured in miles per hour, and the maximum speed measured in kilometers per hour. These two attributes are therefore almost linearly dependent, up to only small differences introduced by rounding off to the nearest mph or kph. Thus, the data really lies approximately on an $n - 1$ dimensional subspace. How can we automatically detect, and perhaps remove, this redundancy?

For a less contrived example, consider a dataset resulting from a survey of pilots for radio-controlled helicopters, where $x_1^{(i)}$ is a measure of the piloting skill of pilot i, and $x_2^{(i)}$ captures how much he/she enjoys flying. Because RC helicopters are very difficult to fly, only the most committed students, ones that truly enjoy flying, become good pilots. So, the two attributes x_1 and x_2 are strongly correlated. Indeed, we might posit that that the data actually likes along some diagonal axis (the u_1 direction) capturing the intrinsic piloting "karma" of a person, with only a small amount of noise lying off this axis. (See figure.) How can we automatically compute this u_1 direction?
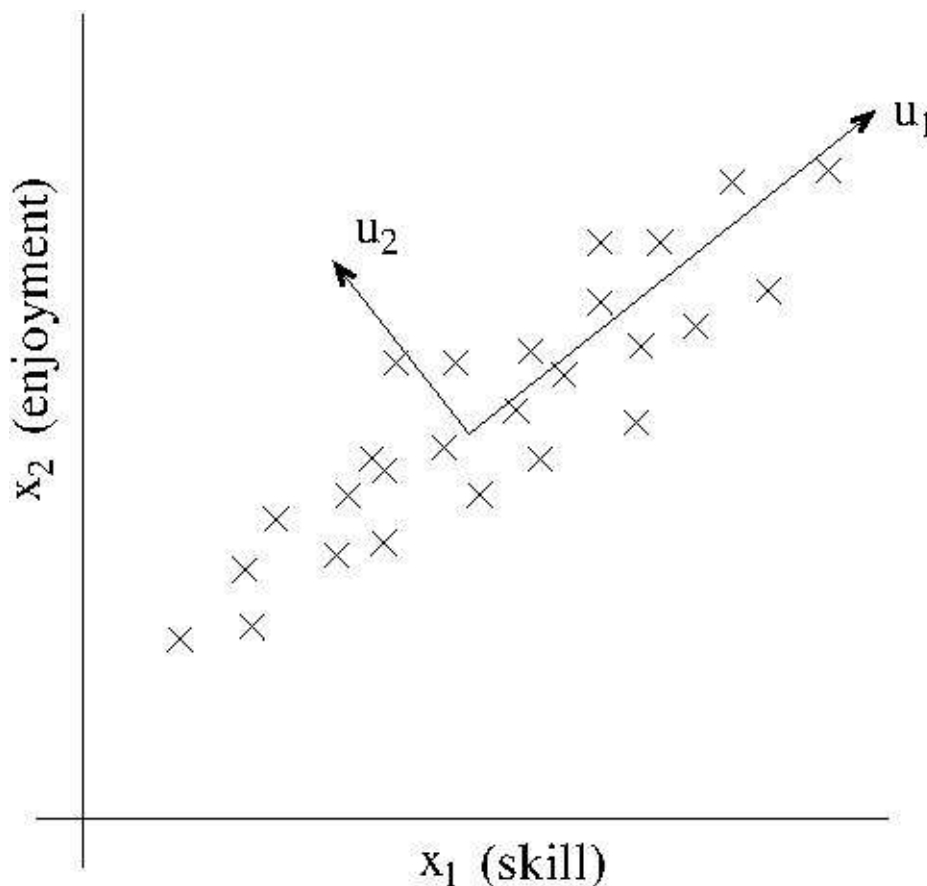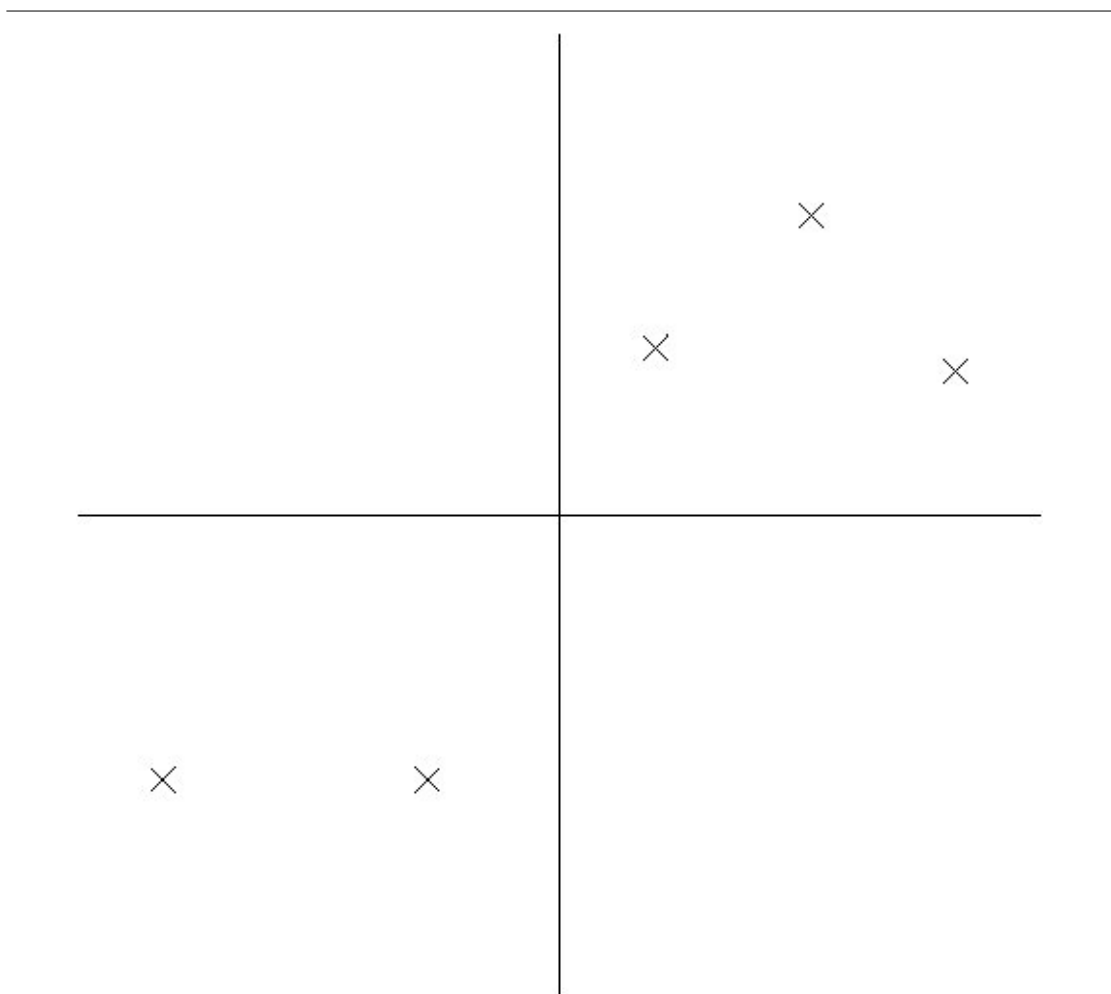

Figure 2.43

We will shortly develop the PCA algorithm. But prior to running PCA per se, typically we first pre-process the data to normalize its mean and variance, as follows:

1. Let $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$.
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$.
3. Let $\sigma_j^2 = \frac{1}{m} \sum_i \left(x_j^{(i)} \right)^2$
4. Replace each $x_j^{(i)}$ with $x_j^{(i)} / \sigma_j$.

Steps (1-2) zero out the mean of the data, and may be omitted for data known to have zero mean (for instance, time series corresponding to speech or other acoustic signals). Steps (3-4) rescale each coordinate to have unit variance, which ensures that different attributes are all treated on the same "scale." For instance, if x_1 was cars' maximum speed in mph (taking values in the high tens or low hundreds) and x_2 were the number of seats (taking values around 2-4), then this renormalization rescales the different attributes to make them more comparable. Steps (3-4) may be omitted if we had apriori knowledge that the different attributes are all on the same scale. One example of this is if each data point represented a grayscale image, and each $x_j^{(i)}$ took a value in $\{0, 1, ..., 255\}$ corresponding to the intensity value of pixel j in image i.

Now, having carried out the normalization, how do we compute the "major axis of variation" u—that is, the direction on which the data approximately lies? One way to pose this problem is as finding the unit vector u so that when the data is projected onto the direction corresponding to u, the variance of the projected data is maximized. Intuitively, the data starts off with some amount of variance/information in it. We would like to choose a direction u so that if we were to approximate the data as lying in the direction/subspace corresponding to u, as much as possible of this variance is still retained.

Figure 2.44

Now, suppose we pick u to correspond the the direction shown in the figure below. The circles denote the projections of the original data onto this line.
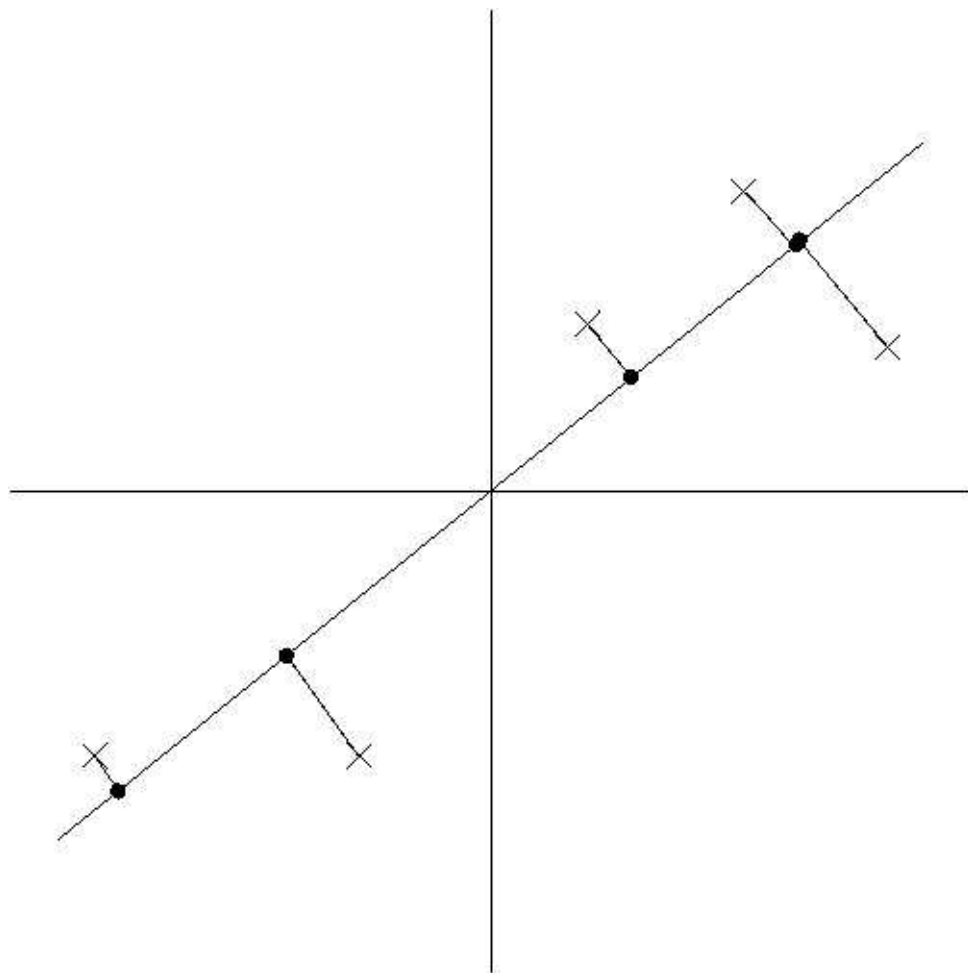
Figure 2.45

We see that the projected data still has a fairly large variance, and the points tend to be far from zero. In contrast, suppose had instead picked the following direction:
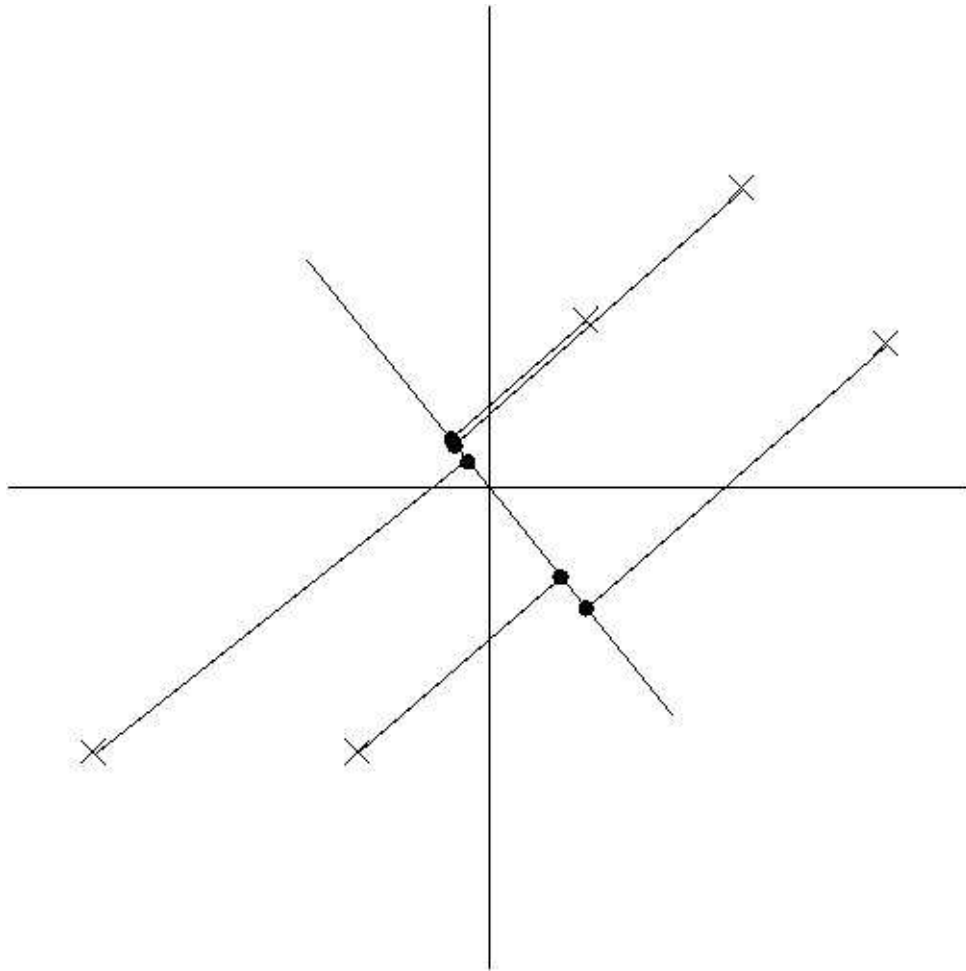
Figure 2.46

Here, the projections have a significantly smaller variance, and are much closer to the origin.

We would like to automatically select the direction u corresponding to the first of the two figures shown above. To formalize this, note that given a unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e., if $x^{(i)}$ is a point in our dataset (one of the crosses in the plot), then its projection onto u (the corresponding circle in the figure) is distance $x^T u$ from the origin. Hence, to maximize the variance of the projections, we would like to choose a unit-length u so as to maximize:

$$
\begin{aligned}
\frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)T} u \right)^2 &= \frac{1}{m} \sum_{i=1}^{m} u^T x^{(i)} x^{(i)T} u \\
&= u^T \left(\frac{1}{m} \sum_{i=1}^{m} x^{(i)} x^{(i)T} \right) u.
\end{aligned}
\tag{2.245}
$$

We easily recognize that the maximizing this subject to $||u||_2 = 1$ gives the principal eigenvector of $\Sigma = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} x^{(i)T}$, which is just the empirical covariance matrix of the data (assuming it has zero mean).[38]

---

[38] If you haven't seen this before, try using the method of Lagrange multipliers to maximize $u^T \Sigma u$ subject to that $u^T u = 1$.

To summarize, we have found that if we wish to find a 1-dimensional subspace with with to approximate the data, we should choose u to be the principal eigenvector of Σ. More generally, if we wish to project our data into a k-dimensional subspace $(k < n)$, we should choose $u_1, ..., u_k$ to be the top k eigenvectors of Σ. The u_i's now form a new, orthogonal basis for the data.[39]

Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ u_2^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k. \tag{2.246}$$

Thus, whereas $x^{(i)} \in \mathbb{R}^n$, the vector $y^{(i)}$ now gives a lower, k-dimensional, approximation/representation for $x^{(i)}$. PCA is therefore also referred to as a **dimensionality reduction** algorithm. The vectors $u_1, ..., u_k$ are called the first k **principal components** of the data.

Remark. Although we have shown it formally only for the case of $k = 1$, using well-known properties of eigenvectors it is straightforward to show that of all possible orthogonal bases $u_1, ..., u_k$, the one that we have chosen maximizes $\sum_i ||y^{(i)}||_2^2$. Thus, our choice of a basis preserves as much variability as possible in the original data.

In problem set 4, you will see that PCA can also be derived by picking the basis that minimizes the approximation error arising from projecting the data onto the k-dimensional subspace spanned by them.

PCA has many applications; we will close our discussion with a few examples. First, compression—representing $x^{(i)}$'s with lower dimension $y^{(i)}$'s—is an obvious application. If we reduce high dimensional data to $k = 2$ or 3 dimensions, then we can also plot the $y^{(i)}$'s to visualize the data. For instance, if we were to reduce our automobiles data to 2 dimensions, then we can plot it (one point in our plot would correspond to one car type, say) to see what cars are similar to each other and what groups of cars may cluster together.

Another standard application is to preprocess a dataset to reduce its dimension before running a supervised learning learning algorithm with the $x^{(i)}$'s as inputs. Apart from computational benefits, reducing the data's dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting (e.g., linear classifiers over lower dimensional input spaces will have smaller VC dimension).

Lastly, as in our RC pilot example, we can also view PCA as a noise reduction algorithm. In our example it estimates the intrinsic "piloting karma" from the noisy measures of piloting skill and enjoyment. In class, we also saw the application of this idea to face images, resulting in **eigenfaces** method. Here, each point $x^{(i)} \in \mathbb{R}^{100 \times 100}$ was a 10000 dimensional vector, with each coordinate corresponding to a pixel intensity value in a 100x100 image of a face. Using PCA, we represent each image $x^{(i)}$ with a much lower-dimensional $y^{(i)}$. In doing so, we hope that the principal components we found retain the interesting, systematic variations between faces that capture what a person really looks like, but not the "noise" in the images introduced by minor lighting variations, slightly different imaging conditions, and so on. We then measure distances between faces i and j by working in the reduced dimension, and computing $||y^{(i)} - y^{(j)}||_2$. This resulted in a surprisingly good face-matching and retrieval algorithm.

2.12 Machine Learning Lecture 11 Course Notes[40]

2.12.1 Independent Components Analysis

Our next topic is Independent Components Analysis (ICA). Similar to PCA, this will find a new basis in which to represent our data. However, the goal is very different.

You should be able to show that $\Sigma u = \lambda u$, for some λ, which implies u is an eigenvector of Σ, with eigenvalue λ.

[39]Because Σ is symmetric, the u_i's will (or always can be chosen to be) orthogonal to each other.

[40]This content is available online at <http://cnx.org/content/m45920/1.3/>.

As a motivating example, consider the "cocktail party problem." Here, n speakers are speaking simultaneously at a party, and any microphone placed in the room records only an overlapping combination of the n speakers' voices. But let's say we have n different microphones placed in the room, and because each microphone is a different distance from each of the speakers, it records a different combination of the speakers' voices. Using these microphone recordings, can we separate out the original n speakers' speech signals?

To formalize this problem, we imagine that there is some data $s \in \mathbb{R}^n$ that is generated via n independent sources. What we observe is

$$x = As, \tag{2.247}$$

where A is an unknown square matrix called the **mixing matrix**. Repeated observations gives us a dataset $\{x^{(i)}; i = 1, ..., m\}$, and our goal is to recover the sources $s^{(i)}$ that had generated our data ($x^{(i)} = As^{(i)}$).

In our cocktail party problem, $s^{(i)}$ is an n-dimensional vector, and $s_j^{(i)}$ is the sound that speaker j was uttering at time i. Also, $x^{(i)}$ in an n-dimensional vector, and $x_j^{(i)}$ is the acoustic reading recorded by microphone j at time i.

Let $W = A^{-1}$ be the **unmixing matrix.** Our goal is to find W, so that given our microphone recordings $x^{(i)}$, we can recover the sources by computing $s^{(i)} = Wx^{(i)}$. For notational convenience, we also let w_i^T denote the i-th row of W, so that

$$W = \begin{bmatrix} --- & w_1^T & --- \\ & \vdots & \\ --- & w_n^T & --- \end{bmatrix}. \tag{2.248}$$

Thus, $w_i \in \mathbb{R}^n$, and the j-th source can be recovered by computing $s_j^{(i)} = w_j^T x^{(i)}$.

2.12.2 ICA ambiguities

To what degree can $W = A^{-1}$ be recovered? If we have no prior knowledge about the sources and the mixing matrix, it is not hard to see that there are some inherent ambiguities in A that are impossible to recover, given only the $x^{(i)}$'s.

Specifically, let P be any n-by-n permutation matrix. This means that each row and each column of P has exactly one "1." Here're some examples of permutation matrices:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \quad P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{2.249}$$

If z is a vector, then Pz is another vector that's contains a permuted version of z's coordinates. Given only the $x^{(i)}$'s, there will be no way to distinguish between W and PW. Specifically, the permutation of the original sources is ambiguous, which should be no surprise. Fortunately, this does not matter for most applications.

Further, there is no way to recover the correct scaling of the w_i's. For instance, if A were replaced with $2A$, and every $s^{(i)}$ were replaced with $(0.5) s^{(i)}$, then our observed $x^{(i)} = 2A \cdot (0.5) s^{(i)}$ would still be the same. More broadly, if a single column of A were scaled by a factor of α, and the corresponding source were scaled by a factor of $1/\alpha$, then there is again no way, given only the $x^{(i)}$'s to determine that this had happened. Thus, we cannot recover the "correct" scaling of the sources. However, for the applications that we are concerned with—including the cocktail party problem—this ambiguity also does not matter. Specifically, scaling a speaker's speech signal $s_j^{(i)}$ by some positive factor α affects only the volume of that speaker's speech. Also, sign changes do not matter, and $s_j^{(i)}$ and $-s_j^{(i)}$ sound identical when played on a speaker. Thus, if the w_i found by an algorithm is scaled by any non-zero real number, the corresponding

recovered source $s_i = w_i^T x$ will be scaled by the same factor; but this usually does not matter. (These comments also apply to ICA for the brain/MEG data that we talked about in class.)

Are these the only sources of ambiguity in ICA? It turns out that they are, so long as the sources s_i are *non-Gaussian*. To see what the difficulty is with Gaussian data, consider an example in which $n = 2$, and $s \sim \mathcal{N}(0, I)$. Here, I is the 2x2 identity matrix. Note that the contours of the density of the standard normal distribution $\mathcal{N}(0, I)$ are circles centered on the origin, and the density is rotationally symmetric.

Now, suppose we observe some $x = As$, where A is our mixing matrix. The distribution of x will also be Gaussian, with zero mean and covariance $E\left[xx^T\right] = E\left[Ass^T A^T\right] = AA^T$. Now, let R be an arbitrary orthogonal (less formally, a rotation/reflection) matrix, so that $RR^T = R^T R = I$, and let $A^{'} = AR$. Then if the data had been mixed according to $A^{'}$ instead of A, we would have instead observed $x^{'} = A^{'}s$. The distribution of $x^{'}$ is also Gaussian, with zero mean and covariance $E\left[x^{'}\left(x^{'}\right)^T\right] = E\left[A^{'}ss^T\left(A^{'}\right)^T\right] = E\left[ARss^T(AR)^T\right] = ARR^T A^T = AA^T$. Hence, whether the mixing matrix is A or $A^{'}$, we would observe data from a $\mathcal{N}\left(0, AA^T\right)$ distribution. Thus, there is no way to tell if the sources were mixed using A and $A^{'}$. So, there is an arbitrary rotational component in the mixing matrix that cannot be determined from the data, and we cannot recover the original sources.

Our argument above was based on the fact that the multivariate standard normal distribution is rotationally symmetric. Despite the bleak picture that this paints for ICA on Gaussian data, it turns out that, so long as the data is *not* Gaussian, it is possible, given enough data, to recover the n independent sources.

2.12.3 Densities and linear transformations

Before moving on to derive the ICA algorithm proper, we first digress briefly to talk about the effect of linear transformations on densities.

Suppose we have a random variable s drawn according to some density $p_s(s)$. For simplicity, let us say for now that $s \in \mathbb{R}$ is a real number. Now, let the random variable x be defined according to $x = As$ (here, $x \in \mathbb{R}, A \in \mathbb{R}$). Let p_x be the density of x. What is p_x?

Let $W = A^{-1}$. To calculate the "probability" of a particular value of x, it is tempting to compute $s = Wx$, then evaluate p_s at that point, and conclude that "$p_x(x) = p_s(Wx)$." However, *this is incorrect*. For example, let $s \sim \text{Uniform}[0, 1]$, so that s's density is $p_s(s) = 1\{0 \le s \le 1\}$. Now, let $A = 2$, so that $x = 2s$. Clearly, x is distributed uniformly in the interval $[0, 2]$. Thus, its density is given by $p_x(x) = (0.5)\,1\{0 \le x \le 2\}$. This does not equal $p_s(Wx)$, where $W = 0.5 = A^{-1}$. Instead, the correct formula is $p_x(x) = p_s(Wx)\,|W|$.

More generally, if s is a vector-valued distribution with density p_s, and $x = As$ for a square, invertible matrix A, then the density of x is given by

$$p_x(x) = p_s(Wx) \cdot |W|, \tag{2.250}$$

where $W = A^{-1}$.

Remark. If you've seen the result that A maps $[0, 1]^n$ to a set of volume $|A|$, then here's another way to remember the formula for p_x given above, that also generalizes our previous 1-dimensional example. Specifically, let $A \in \mathbb{R}^{n \times n}$ be given, and let $W = A^{-1}$ as usual. Also let $C_1 = [0, 1]^n$ be the n-dimensional hypercube, and define $C_2 = \{As : s \in C_1\} \subseteq \mathbb{R}^n$ to be the image of C_1 under the mapping given by A. Then it is a standard result in linear algebra (and, indeed, one of the ways of defining determinants) that the volume of C_2 is given by $|A|$. Now, suppose s is uniformly distributed in $[0, 1]^n$, so its density is $p_s(s) = 1\{s \in C_1\}$. Then clearly x will be uniformly distributed in C_2. Its density is therefore found to be $p_x(x) = 1\{x \in C_2\}/\text{vol}(C_2)$ (since it must integrate over C_2 to 1). But using the fact that the determinant of the inverse of a matrix is just the inverse of the determinant, we have $1/\text{vol}(C_2) = 1/|A| = |A^{-1}| = |W|$. Thus, $p_x(x) = 1\{x \in C_2\}|W| = 1\{Wx \in C_1\}|W| = p_s(Wx)|W|$.

2.12.4 ICA algorithm

We are now ready to derive an ICA algorithm. The algorithm we describe is due to Bell and Sejnowski, and the interpretation we give will be of their algorithm as a method for maximum likelihood estimation. (This is different from their original interpretation, which involved a complicated idea called the infomax principal, that is no longer necessary in the derivation given the modern understanding of ICA.)

We suppose that the distribution of each source s_i is given by a density p_s, and that the joint distribution of the sources s is given by

$$p\left(s\right) = \prod_{i=1}^{n} p_s\left(s_i\right). \tag{2.251}$$

Note that by modeling the joint distribution as a product of the marginal, we capture the assumption that the sources are independent. Using our formulas from the previous section, this implies the following density on $x = As = W^{-1}s$:

$$p\left(x\right) = \prod_{i=1}^{n} p_s\left(w_i^T x\right) \cdot |W|. \tag{2.252}$$

All that remains is to specify a density for the individual sources p_s.

Recall that, given a real-valued random variable z, its cumulative distribution function (cdf) F is defined by $F\left(z_0\right) = P\left(z \leq z_0\right) = \int_{-\infty}^{z_0} p_z\left(z\right) dz$. Also, the density of z can be found from the cdf by taking its derivative: $p_z\left(z\right) = F^{'}\left(z\right)$.

Thus, to specify a density for the s_i's, all we need to do is to specify some cdf for it. A cdf has to be a monotonic function that increases from zero to one. Following our previous discussion, we cannot choose the cdf to be the cdf of the Gaussian, as ICA doesn't work on Gaussian data. What we'll choose instead for the cdf, as a reasonable "default" function that slowly increases from 0 to 1, is the sigmoid function $g\left(s\right) = 1/\left(1 + e^{-s}\right)$. Hence, $p_s\left(s\right) = g^{'}\left(s\right)$.[41]

The square matrix W is the parameter in our model. Given a training set $\{x^{(i)}; i = 1, ..., m\}$, the log likelihood is given by

$$\ell\left(W\right) = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \log g^{'}\left(w_j^T x^{(i)}\right) + \log |W| \right). \tag{2.253}$$

We would like to maximize this in terms W. By taking derivatives and using the fact (from the first set of notes) that $\nabla_W |W| = |W|\left(W^{-1}\right)^T$, we easily derive a stochastic gradient ascent learning rule. For a training example $x^{(i)}$, the update rule is:

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g\left(w_1^T x^{(i)}\right) \\ 1 - 2g\left(w_2^T x^{(i)}\right) \\ \vdots \\ 1 - 2g\left(w_n^T x^{(i)}\right) \end{bmatrix} x^{(i)T} + \left(W^T\right)^{-1} \right), \tag{2.254}$$

where α is the learning rate.

After the algorithm converges, we then compute $s^{(i)} = W x^{(i)}$ to recover the original sources.

---

[41] If you have prior knowledge that the sources' densities take a certain form, then it is a good idea to substitute that in here. But in the absence of such knowledge, the sigmoid function can be thought of as a reasonable default that seems to work well for many problems. Also, the presentation here assumes that either the data $x^{(i)}$ has been preprocessed to have zero mean, or that it can naturally be expected to have zero mean (such as acoustic signals). This is necessary because our assumption that $p_s\left(s\right) = g^{'}\left(s\right)$ implies $E\left[s\right] = 0$ (the derivative of the logistic function is a symmetric function, and hence gives a density corresponding to a random variable with zero mean), which implies $E\left[x\right] = E\left[As\right] = 0$.

Remark. When writing down the likelihood of the data, we implicity assumed that the $x^{(i)}$'s were independent of each other (for different values of i; note this issue is different from whether the different coordinates of $x^{(i)}$ are independent), so that the likelihood of the training set was given by $\prod_i p\left(x^{(i)}; W\right)$. This assumption is clearly incorrect for speech data and other time series where the $x^{(i)}$'s are dependent, but it can be shown that having correlated training examples will not hurt the performance of the algorithm if we have sufficient data. But, for problems where successive training examples are correlated, when implementing stochastic gradient ascent, it also sometimes helps accelerate convergence if we visit training examples in a randomly permuted order. (I.e., run stochastic gradient ascent on a randomly shuffled copy of the training set.)

2.13 Machine Learning Lecture 12 Course Notes[42]

2.13.1 Reinforcement Learning and Control

We now begin our study of reinforcement learning and adaptive control.

In supervised learning, we saw algorithms that tried to make their outputs mimic the labels y given in the training set. In that setting, the labels gave an unambiguous "right answer" for each of the inputs x. In contrast, for many sequential decision making and control problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the "correct" actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

In the reinforcement learning framework, we will instead provide our algorithms only a reward function, which indicates to the learning agent when it is doing well, and when it is doing poorly. In the four-legged walking example, the reward function might give the robot positive rewards for moving forwards, and negative rewards for either moving backwards or falling over. It will then be the learning algorithm's job to figure out how to choose actions over time so as to obtain large rewards.

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web-page indexing. Our study of reinforcement learning will begin with a definition of the **Markov decision processes (MDP)**, which provides the formalism in which RL problems are usually posed.

2.13.2 Markov decision processes

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter.)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- P_{sa} are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briefly, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \mapsto \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \mapsto \mathbb{R}$).

The dynamics of an MDP proceeds as follows: We start in some state s_0, and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$. Then, we get to pick another action a_1. As a result of

---

[42]This content is available online at <http://cnx.org/content/m45954/1.4/>.

this action, the state transitions again, now to some $s_2 \sim P_{s_1 a_1}$. We then pick a_2, and so on.... Pictorially, we can represent this process as follows:

$$s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} s_2 \overset{a_2}{\to} s_3 \overset{a_3}{\to} ... \tag{2.255}$$

Upon visiting the sequence of states $s_0, s_1, ...$ with actions $a_0, a_1, ...$, our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots . \tag{2.256}$$

Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots . \tag{2.257}$$

For most of our development, we will use the simpler state-rewards $R(s)$, though the generalization to state-action rewards $R(s, a)$ offers no special difficulties.

Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots\right] \tag{2.258}$$

Note that the reward at timestep t is **discounted** by a factor of γ^t. Thus, to make this expectation large, we would like to accrue positive rewards as soon as possible (and postpone negative rewards as long as possible). In economic applications where $R(\cdot)$ is the amount of money made, γ also has a natural interpretation in terms of the interest rate (where a dollar today is worth more than a dollar tomorrow).

A **policy** is any function $\pi : S \mapsto A$ mapping from the states to the actions. We say that we are **executing** some policy π if, whenever we are in state s, we take action $a = \pi(s)$. We also define the **value function** for a policy π according to

$$V^\pi(s) = E\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \,\middle|\, s_0 = s, \pi\right]. \tag{2.259}$$

$V^\pi(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.[43]

Given a fixed policy π, its value function V^π satisfies the **Bellman equations**:

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}\left(s'\right) V^\pi\left(s'\right). \tag{2.260}$$

This says that the expected sum of discounted rewards $V^\pi(s)$ for starting in s consists of two terms: First, the **immediate reward** $R(s)$ that we get rightaway simply for starting in state s, and second, the expected sum of future discounted rewards. Examining the second term in more detail, we see that the summation term above can be rewritten $E_{s' \sim P_{s\pi(s)}}\left[V^\pi\left(s'\right)\right]$. This is the expected sum of discounted rewards for starting in state s', where s' is distributed according $P_{s\pi(s)}$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s. Thus, the second term above gives the expected sum of discounted rewards obtained *after* the first step in the MDP.

Bellman's equations can be used to efficiently solve for V^π. Specifically, in a finite-state MDP ($|S| < \infty$), we can write down one such equation for $V^\pi(s)$ for every state s. This gives us a set of $|S|$ linear equations in $|S|$ variables (the unknown $V^\pi(s)$'s, one for each state), which can be efficiently solved for the $V^\pi(s)$'s.

We also define the **optimal value function** according to

$$V^*(s) = \max_\pi V^\pi(s). \tag{2.261}$$

---

[43]This notation in which we condition on π isn't technically correct because π isn't a random variable, but this is quite standard in the literature.

In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy. There is also a version of Bellman's equations for the optimal value function:

$$V^* \left(s \right) = R \left(s \right) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa} \left(s' \right) V^* \left(s' \right).$$
(2.262)

The first term above is the immediate reward as before. The second term is the maximum over all actions a of the expected future sum of discounted rewards we'll get upon after action a. You should make sure you understand this equation and see why it makes sense.

We also define a policy $\pi^* : S \mapsto A$ as follows:

$$\pi^* \left(s \right) = arg \max_{a \in A} \sum_{s' \in S} P_{sa} \left(s' \right) V^* \left(s' \right).$$
(2.263)

Note that $\pi^* \left(s \right)$ gives the action a that attains the maximum in the "max" in Equation (2.262).

It is a fact that for every state s and every policy π, we have

$$V^* \left(s \right) = V^{\pi^*} \left(s \right) \geq V^{\pi} \left(s \right).$$
(2.264)

The first equality says that the V^{π^*}, the value function for π^*, is equal to the optimal value function V^* for every state s. Further, the inequality above says that π^*'s value is at least a large as the value of any other other policy. In other words, π^* as defined in Equation (2.263) is the optimal policy.

Note that π^* has the interesting property that it is the optimal policy for *all* states s. Specifically, it is not the case that if we were starting in some state s then there'd be some optimal policy for that state, and if we were starting in some other state s' then there'd be some other policy that's optimal policy for s'. Specifically, the same policy π^* attains the maximum in Equation (2.261) for *all* states s. This means that we can use the same policy π^* no matter what the initial state of our MDP is.

2.13.3 Value iteration and policy iteration

We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces ($|S| < \infty$, $|A| < \infty$).

The first algorithm, **value iteration**, is as follows:

1. For each state s, initialize $V \left(s \right) := 0$.
2. Repeat until convergence {

   - For every state, update $V \left(s \right) := R \left(s \right) + max_{a \in A} \gamma \sum_{s'} P_{sa} \left(s' \right) V \left(s' \right)$.

3. }

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations (2.262).

There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for $V \left(s \right)$ for every state s, and then overwrite all the old values with the new values. This is called a **synchronous** update. In this case, the algorithm can be viewed as implementing a "Bellman backup operator" that takes a current estimate of the value function, and maps it to a new estimate. (See homework problem for details.) Alternatively, we can also perform **asynchronous** updates. Here, we would loop over the states (in some order), updating the values one at a time.

Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can then use Equation (2.263) to find the optimal policy.

Apart from value iteration, there is a second standard algorithm for finding an optimal policy for an MDP. The **policy iteration** algorithm proceeds as follows:

1. Initialize π randomly.

2. Repeat until convergence {

    a. Let $V := V^\pi$.

    b. For each state s, let $\pi(s) := argmax_{a \in A} \sum_{s'} P_{sa}(s') V(s')$.

3. }

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is **greedy with respect to** V.) Note that step (a) can be done via solving Bellman's equations as described earlier, which in the case of a fixed policy, is just a set of $|S|$ linear equations in $|S|$ variables.

After at most a finite number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn't currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V^π explicitly would involve solving a large system of linear equations, and could be difficult. In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.

2.13.4 Learning a model for an MDP

So far, we have discussed MDPs and algorithms for MDPs assuming that the state transition probabilities and rewards are known. In many realistic problems, we are not given state transition probabilities and rewards explicitly, but must instead estimate them from data. (Usually, S, A and γ are known.)

For example, suppose that, for the inverted pendulum problem (see problem set 4), we had a number of trials in the MDP, that proceeded as follows:

$$
\begin{aligned}
s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \ldots \\
s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \ldots
\end{aligned}
\tag{2.265}
$$

$$\ldots$$

Here, $s_i^{(j)}$ is the state we were at time i of trial j, and $a_i^{(j)}$ is the corresponding action that was taken from that state. In practice, each of the trials above might be run until the MDP terminates (such as if the pole falls over in the inverted pendulum problem), or it might be run for some large but finite number of timesteps.

Given this "experience" in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:

$$
P_{sa}(s') \frac{\#\text{times took we action}\, a\, \text{in state}\, s\, \text{and got to}\, s'}{\#\text{times we took action}\, a\, \text{in state}\, s}
\tag{2.266}
$$

Or, if the ratio above is "0/0"—corresponding to the case of never having taken action a in state s before—the we might simply estimate $P_{sa}(s')$ to be $1/|S|$. (I.e., estimate P_{sa} to be the uniform distribution over all states.)

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of (2.266), then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of P_{sa}.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilities and rewards. For example, putting together model

learning and value iteration, here is one possible algorithm for learning in an MDP with unknown state transition probabilities:

1. Initialize π randomly.
2. Repeat {

    a. Execute π in the MDP for some number of trials.

    b. Using the accumulated experience in the MDP, update our estimates for P_{sa} (and R, if applicable).

    c. Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function V.

    d. Update π to be the greedy policy with respect to V.

3. }

can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with $V = 0$, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

2.13.5 Continuous state MDPs

So far, we've focused our attention on MDPs with a finite number of states. We now discuss algorithms for MDPs that may have an infinite number of states. For example, for a car, we might represent the state as $\left(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}\right)$, comprising its position (x, y); orientation θ; velocity in the x and y directions $\dot{x}$ and $\dot{y}$; and angular velocity $\dot{\theta}$. Hence, $S = \mathbb{R}^6$ is an infinite set of states, because there is an infinite number of possible positions and orientations for the car.[44] Similarly, the inverted pendulum you saw in PS4 has states $\left(x, \theta, \dot{x}, \dot{\theta}\right)$, where θ is the angle of the pole. And, a helicopter flying in 3d space has states of the form $\left(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}\right)$, where here the roll ϕ, pitch θ, and yaw ψ angles specify the 3d orientation of the helicopter.

In this section, we will consider settings where the state space is $S = \mathbb{R}^n$, and describe ways for solving such MDPs.

2.13.5.1 Discretization

Perhaps the simplest way to solve a continuous-state MDP is to discretize the state space, and then to use an algorithm like value iteration or policy iteration, as described previously.

For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:

---

[44]Technically, θ is an orientation and so the range of θ is better written $\theta \in [-\pi, \pi)$ than $\theta \in \mathbb{R}$; but for our purposes, this distinction is not important.
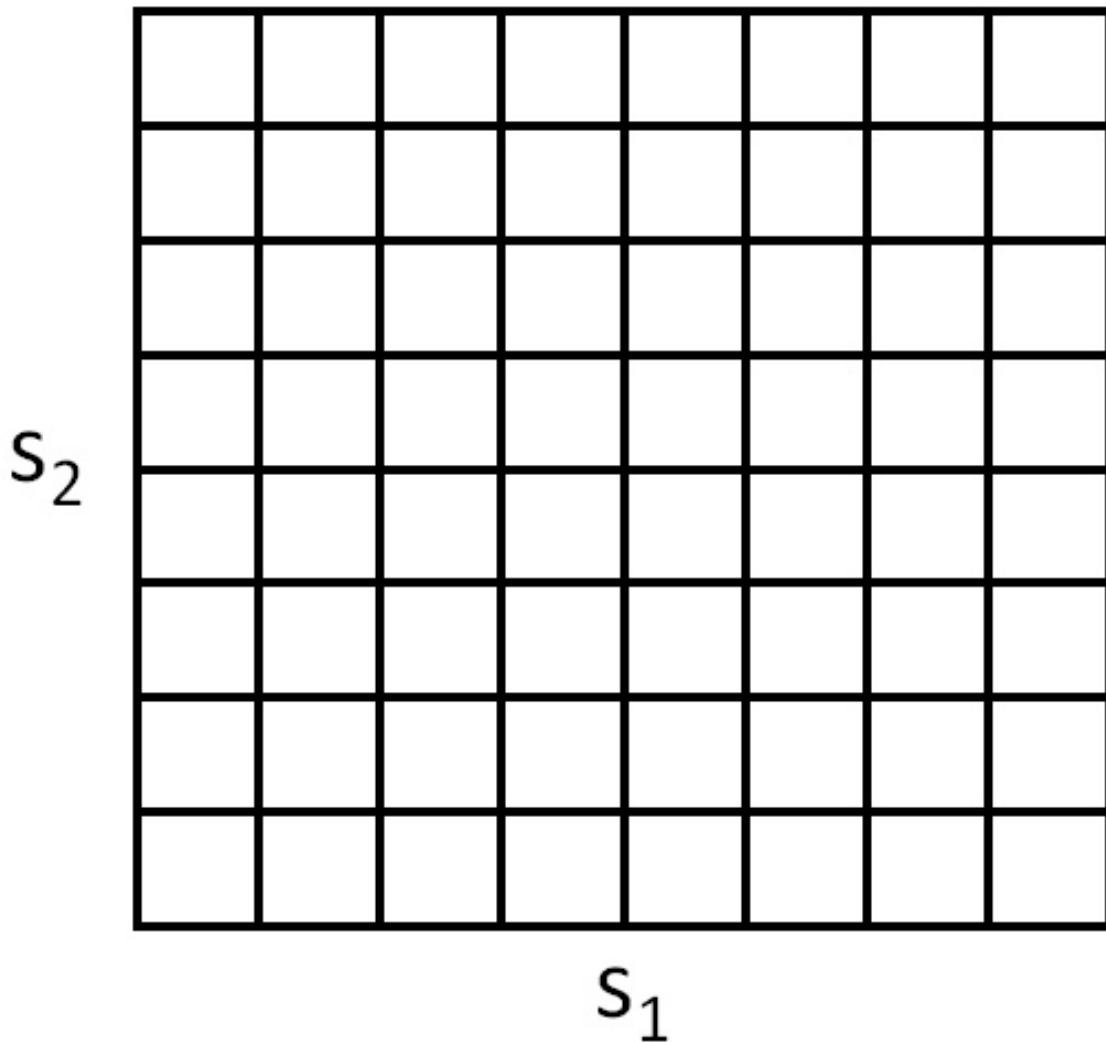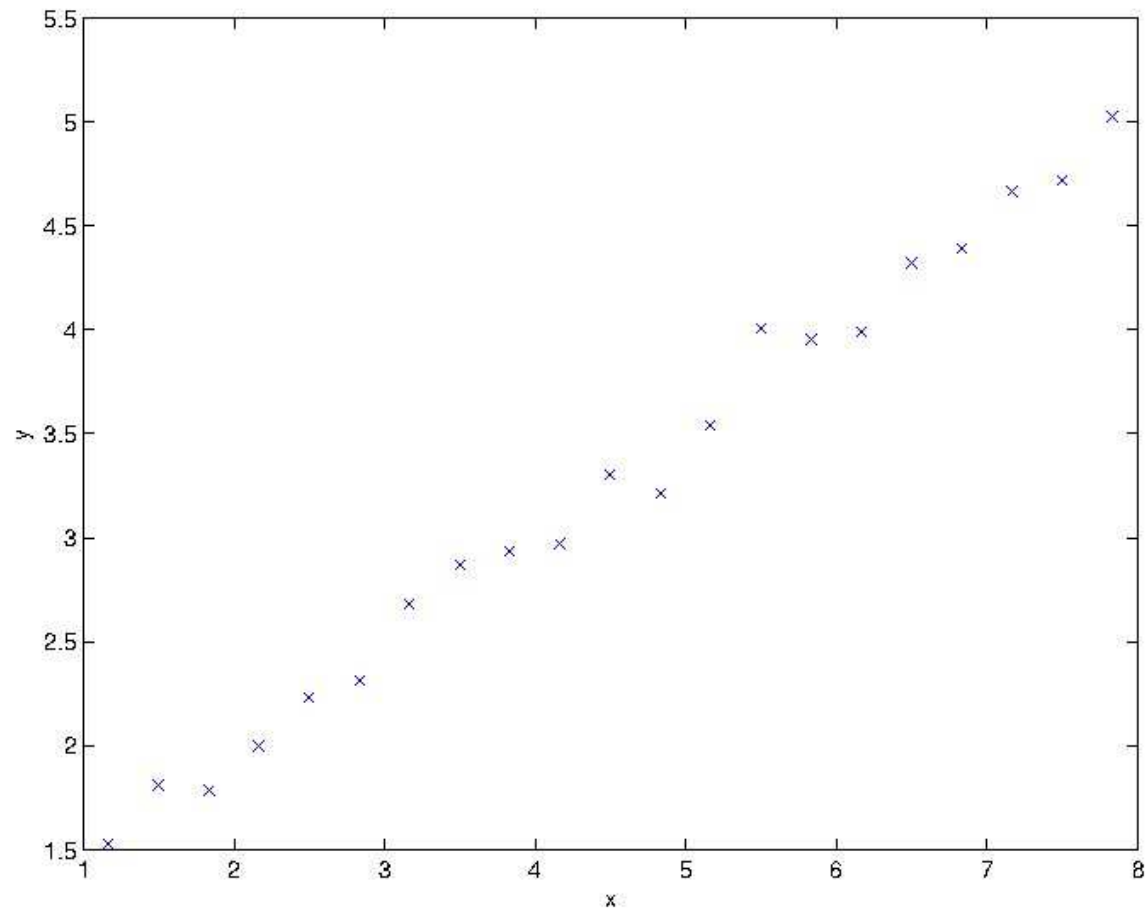
Figure 2.47

Here, each grid cell represents a separate discrete state $\overline{s}$. We can then approximate the continuous-state MDP via a discrete-state one $\left(\overline{S}, A, \{P_{\overline{s}a}\}, \gamma, R\right)$, where $\overline{S}$ is the set of discrete states, $\{P_{\overline{s}a}\}$ are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the $V^*\left(\overline{s}\right)$ and $\pi^*\left(\overline{s}\right)$ in the discrete state MDP $\left(\overline{S}, A, \{P_{\overline{s}a}\}, \gamma, R\right)$. When our actual system is in some continuous-valued state $s \in S$ and we need to pick an action to execute, we compute the corresponding discretized state $\overline{s}$, and execute action $\pi^*\left(\overline{s}\right)$.

two downsides. First, it uses a fairly naive representation for V^* (and π^*). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the gridcells).

To better understand the limitations of such a representation, consider a *supervised learning* problem of fitting a function to this dataset:

Figure 2.48

Clearly, linear regression would do fine on this problem. However, if we instead discretize the x-axis, and then use a representation that is piecewise constant in each of the discretization intervals, then our fit to the data would look like this:
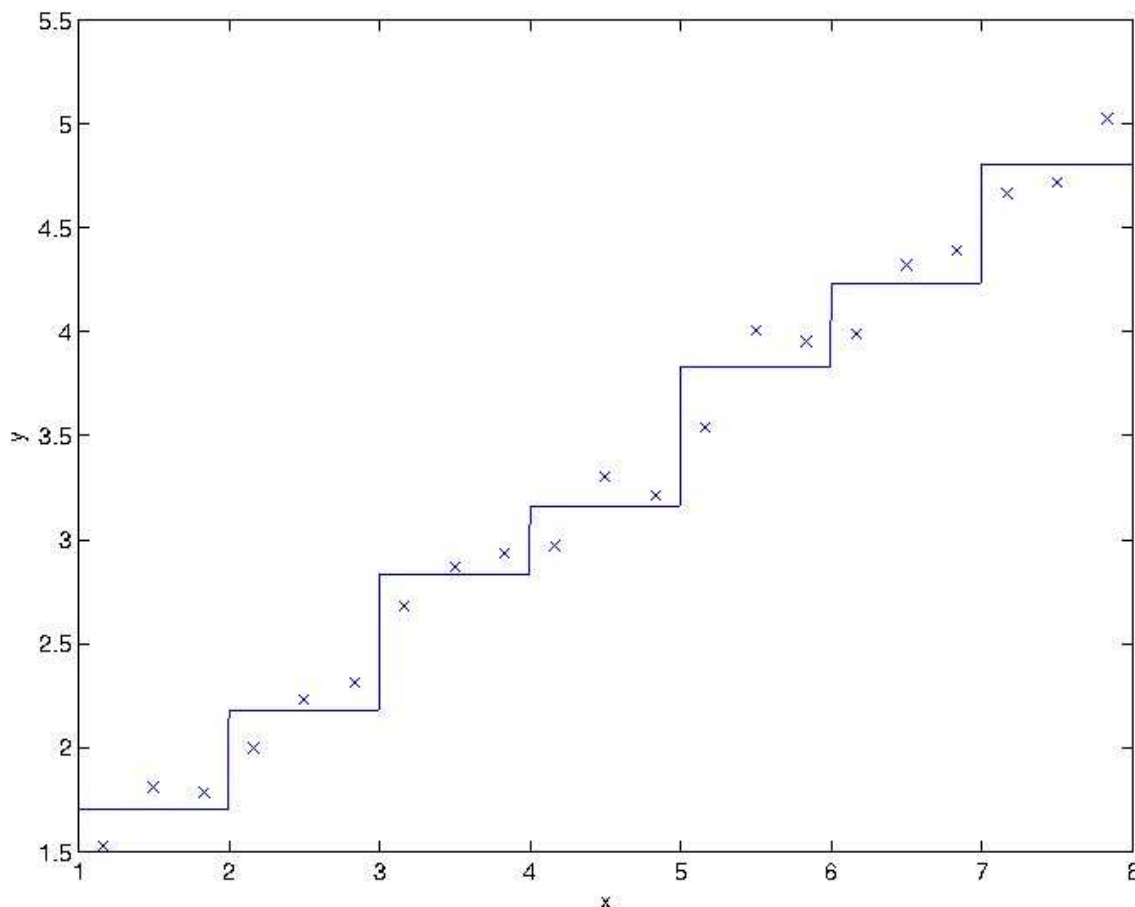
Figure 2.49

This piecewise constant representation just isn't a good representation for many smooth functions. It results in little smoothing over the inputs, and no generalization over the different grid cells. Using this sort of representation, we would also need a very fine discretization (very small grid cells) to get a good approximation.

A second downside of this representation is called the **curse of dimensionality**. Suppose $S = \mathbb{R}^n$, and we discretize each of the n dimensions of the state into k values. Then the total number of discrete states we have is k^n. This grows exponentially quickly in the dimension of the state space n, and thus does not scale well to large problems. For example, with a 10d state, if we discretize each state variable into 100 values, we would have $100^{10} = 10^{20}$ discrete states, which is far too many to represent even on a modern desktop computer.

As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you're extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

2.13.5.2 Value function approximation

We now describe an alternative method for finding policies in continuous-state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, caled value function approximation, has been successfully applied to many RL problems.

2.13.5.2.1 Using a model or simulator

To develop a value function approximation algorithm, we will assume that we have a **model**, or **simulator**, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state s_t and action a_t, and outputs a next-state s_{t+1} sampled according to the state transition probabilities $P_{s_t a_t}$:
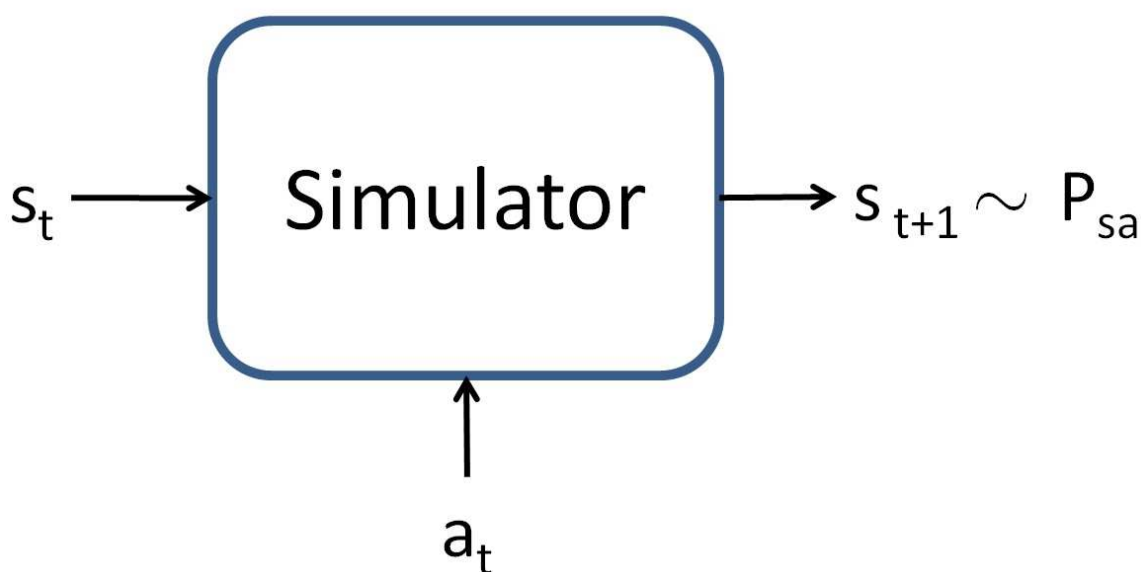
Figure 2.50

simulation. For example, the simulator for the inverted pendulum in PS4 was obtained by using the laws of physics to calculate what position and orientation the cart/pole will be in at time $t+1$, given the current state at time t and the action a taken, assuming that we know all the parameters of the system such as the length of the pole, the mass of the pole, and so on. Alternatively, one can also use an off-the-shelf physics simulation software package which takes as input a complete physical description of a mechanical system, the current state s_t and action a_t, and computes the state s_{t+1} of the system a small fraction of a second into the future.[45]

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute m **trials** in which we repeatedly take actions in an MDP, each trial for T timesteps. This can be done picking actions at random, executing some specific policy, or via some other way of choosing actions.

---

[45]Open Dynamics Engine (http://www.ode.com) is one example of a free/open-source physics simulator that can be used to simulate systems like the inverted pendulum, and that has been a reasonably popular choice among RL researchers.

We would then observe m state sequences like the following:

$$s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} \ldots \xrightarrow{a_{T-1}^{(1)}} s_T^{(1)}$$
$$s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} \ldots \xrightarrow{a_{T-1}^{(2)}} s_T^{(2)} \tag{2.267}$$
$$\ldots$$
$$s_0^{(m)} \xrightarrow{a_0^{(m)}} s_1^{(m)} \xrightarrow{a_1^{(m)}} s_2^{(m)} \xrightarrow{a_2^{(m)}} \ldots \xrightarrow{a_{T-1}^{(m)}} s_T^{(m)}$$

We can then apply a learning algorithm to predict s_{t+1} as a function of s_t and a_t.

For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t, \tag{2.268}$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our m trials, by picking

$$\underset{A,B}{argmin} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \|^2. \tag{2.269}$$

(This corresponds to the maximum likelihood estimate of the parameters.)

Having learned A and B, one option is to build a **deterministic** model, in which given an input s_t and a_t, the output s_{t+1} is exactly determined. Specifically, we always compute s_{t+1} according to Equation (2.268). Alternatively, we may also build a **stochastic** model, in which s_{t+1} is a random function of the inputs, by modelling it as

$$s_{t+1} = As_t + Ba_t + \varepsilon_t, \tag{2.270}$$

where here ε_t is a noise term, usually modeled as $\varepsilon_t \sim \mathcal{N}(0, \Sigma)$. (The covariance matrix Σ can also be estimated from data in a straightforward way.)

Here, we've written the next-state s_{t+1} as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model $s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$, where ϕ_s and ϕ_a are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms, such as locally weighted linear regression, to learn to estimate s_{t+1} as a function of s_t and a_t. These approaches can also be used to build either deterministic or stochastic simulators of an MDP.

2.13.5.2.2 Fitted value iteration

We now describe the **fitted value iteration** algorithm for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space $S = \mathbb{R}^n$, but that the action space A is small and discrete.[46]

Recall that in value iteration, we would like to perform the update

$$
\begin{aligned}
V(s) &:= R(s) + \gamma \underset{a}{max} \int_{s'} P_{sa}\left(s'\right) V\left(s'\right) ds' \\
&= R(s) + \gamma \underset{a}{max} E_{s' \sim P_{sa}}\left[V\left(s'\right)\right]
\end{aligned}
\tag{2.271}
$$

(In "Value iteration and policy iteration" (Section 2.13.3: Value iteration and policy iteration), we had written the value iteration update with a summation $V(s) := R(s) + \gamma max_a \sum_{s'} P_{sa}\left(s'\right) V\left(s'\right)$ rather than

---

[46]In practice, most MDPs have much smaller action spaces than state spaces. E.g., a car has a 6d state space, and a 2d action space (steering and velocity controls); the inverted pendulum has a 4d state space, and a 1d action space; a helicopter has a 12d state space, and a 4d action space. So, discretizing ths set of actions is usually less of a problem than discretizing the state space would have been.

an integral over states; the new notation reflects that we are now working in continuous states rather than discrete states.)

The main idea of fitted value iteration is that we are going to approximately carry out this step, over a finite sample of states $s^{(1)}, ..., s^{(m)}$. Specifically, we will use a supervised learning algorithm—linear regression in our description below—to approximate the value function as a linear or non-linear function of the states:

$$V\left(s\right) = \theta^T \phi\left(s\right). \tag{2.272}$$

Here, ϕ is some appropriate feature mapping of the states.

For each state s in our finite sample of m states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R\left(s\right) + \gamma max_a E_{s' \sim P_{sa}}\left[V\left(s'\right)\right]$ (the right hand side of Equation (2.271)). Then, it will apply a supervised learning algorithm to try to get $V\left(s\right)$ close to $R\left(s\right) + \gamma max_a E_{s' \sim P_{sa}}\left[V\left(s'\right)\right]$ (or, in other words, to try to get $V\left(s\right)$ close to $y^{(i)}$).

In detail, the algorithm is as follows:

1. Randomly sample m states $s^{(1)}, s^{(2)}, ...s^{(m)} \in S$.
2. Initialize $\theta := 0$.
3. Repeat {
4.   a. For $i = 1, ..., m${
            1. For each action $a \in A${
            2.   a. Sample $s_1', ..., s_k' \sim P_{s^{(i)}a}$ (using a model of the MDP).
                   b. Set $q\left(a\right) = \frac{1}{k}\sum_{j=1}^{k} R\left(s^{(i)}\right) + \gamma V\left(s_j'\right)$
                   c.   // Hence, $q\left(a\right)$ is an estimate of $R\left(s^{(i)}\right) + \gamma E_{s' \sim P_{s^{(i)}a}}\left[V\left(s'\right)\right]$.
            3. }
            4. Set $y^{(i)} = max_a q\left(a\right)$.
            5.   // Hence, $y^{(i)}$ is an estimate of $R\left(s^{(i)}\right) + \gamma max_a E_{s' \sim P_{s^{(i)}a}}\left[V\left(s'\right)\right]$.
        b. }
        c. // In the original value iteration algorithm (over discrete states)
        d. // we updated the value function according to $V\left(s^{(i)}\right) := y^{(i)}$.
        e. // In this algorithm, we want $V\left(s^{(i)}\right) \approx y^{(i)}$, which we'll achieve
        f. // using supervised learning (linear regression).
        g. Set $\theta := argmin_\theta \frac{1}{2}\sum_{i=1}^{m}\left(\theta^T \phi\left(s^{(i)}\right) - y^{(i)}\right)^2$
5. }

Above, we had written out fitted value iteration using linear regression as the algorithm to try to make $V\left(s^{(i)}\right)$ close to $y^{(i)}$. That step of the algorithm is completely analogous to a standard supervised learning (regression) problem in which we have a training set $\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), ..., \left(x^{(m)}, y^{(m)}\right)$, and want to learn a function mapping from x to y; the only difference is that here s plays the role of x. Even though our description above used linear regression, clearly other regression algorithms (such as locally weighted linear regression) can also be used.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems. Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting $k = 1$ in the algorithm. This is because the expectation in Equation (2.271) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above, we had to draw k samples, and average to try to approximate that expectation (see the definition of $q\left(a\right)$, in the algorithm pseudo-code).

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

$$argmax_a E_{s' \sim P_{sa}}\left[V\left(s'\right)\right] \tag{2.273}$$

The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample $s_1^{'}, ..., s_k^{'} \sim P_{sa}$ to approximate the expectation. (And again, if the simulator is deterministic, we can set $k = 1$.)

In practice, there're often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \varepsilon_t$, where f is some determinstic function of the states (such as $f(s_t, a_t) = As_t + Ba_t$), and ε is zero-mean Gaussian noise. In this case, we can pick the action given by

$$\underset{a}{argmax} V(f(s, a)). \tag{2.274}$$

In other words, here we are just setting $\varepsilon_t = 0$ (i.e., ignoring the noise in the simulator), and setting $k = 1$. Equivalently, this can be derived from Equation (2.273) using the approximation

$$\begin{aligned} E_{s^{'}}\left[V\left(s^{'}\right)\right] &\approx V\left(E_{s^{'}}\left[s^{'}\right]\right) \\ &= V\left(f\left(s, a\right)\right), \end{aligned} \tag{2.275}$$

where here the expection is over the random $s^{'} \sim P_{sa}$. So long as the noise terms ε_t are small, this will usually be a reasonable approximation.

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above, can be computationally expensive.

Chapter 3

Machine Learning Review Notes[1]

The following modules contains various review notes supporting this machine learning course. Please click the link to download and save the following files.

Linear Algebra Review and Reference	cs229-linalg.pdf[26]
Probability Theory Review	cs229-prob.pdf[27]
Matlab Review	cs229-prob.pdf[28] sigmoid.txt[29] mat-lab_session.txt[30]
Convex Optimization Overview, Part I	cs229-cvxopt.pdf[31]
Convex Optimization Overview, Part II	cs229-cvxopt2.pdf[32]
Hidden Markov Models	cs229-hmm.pdf[33]
Gaussian Processes	cs229-gp.pdf[34] compute_kernel_matrix.txt[35] gp_demo.txt[36] sample_gp_prior.txt[37]

Table 3.1

---

[1]This content is available online at <http://cnx.org/content/m45963/1.3/>.
[26]http://cnx.org/content/m45963/latest/cs229-linalg.pdf
[27]http://cnx.org/content/m45963/latest/cs229-prob.pdf
[28]http://cnx.org/content/m45963/latest/logistic_grad_ascent.txt
[29]http://cnx.org/content/m45963/latest/sigmoid.txt
[30]http://cnx.org/content/m45963/latest/matlab_session.txt
[31]http://cnx.org/content/m45963/latest/cs229-cvxopt.pdf
[32]http://cnx.org/content/m45963/latest/cs229-cvxopt2.pdf
[33]http://cnx.org/content/m45963/latest/cs229-hmm.pdf
[34]http://cnx.org/content/m45963/latest/cs229-gp.pdf
[35]http://cnx.org/content/m45963/latest/compute_kernel_matrix.txt
[36]http://cnx.org/content/m45963/latest/gp_demo.txt
[37]http://cnx.org/content/m45963/latest/sample_gp_prior.txt

Chapter 4

Machine Learning Problem Sets and Solutions[1]

In this module you can find files for the problem sets, the solution sets, and the corresponding data.

Problem Set 1[2]	PS1 Data[3]	PS1 Solution[4]	PS1 Solution Data[5]
Problem Set 2[6]	PS2 Data[7]	PS2 Solution[8]	none
Problem Set 3[9]	PS3 Data[10]	PS3 Solution[11]	PS3 Solution Data[12]
Problem Set 4[13]	PS4 Data[14]	PS4 Solution[15]	PS4 Solution Data[16]

Table 4.1

Practice midterm[17]

---

[1]This content is available online at <http://cnx.org/content/m45957/1.2/>.
[2]http://cnx.org/content/m45957/latest/problemset1.pdf
[3]http://cnx.org/content/m45957/latest/problemset1-data.zip
[4]http://cnx.org/content/m45957/latest/ps1_solution.pdf
[5]http://cnx.org/content/m45957/latest/ps1_solution-data.zip
[6]http://cnx.org/content/m45957/latest/problemset2.pdf
[7]http://cnx.org/content/m45957/latest/problemset2-data.zip
[8]http://cnx.org/content/m45957/latest/ps2_solution.pdf
[9]http://cnx.org/content/m45957/latest/problemset3.pdf
[10]http://cnx.org/content/m45957/latest/problemset3-data.zip
[11]http://cnx.org/content/m45957/latest/ps3_solution.pdf
[12]http://cnx.org/content/m45957/latest/ps3_solution-data.zip
[13]http://cnx.org/content/m45957/latest/problemset4.pdf
[14]http://cnx.org/content/m45957/latest/problemset4-data.zip
[15]http://cnx.org/content/m45957/latest/ps4_solution.pdf
[16]http://cnx.org/content/m45957/latest/ps4_solution-data.zip
[17]http://cnx.org/content/m45957/latest/practice_midterm.pdf

Attributions

Collection: *Machine Learning*
Edited by: Andrew Ng
URL: http://cnx.org/content/col11500/1.4/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Introduction"
By: Andrew Ng
URL: http://cnx.org/content/m45891/1.2/
Pages: 1-2
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Syllabus"
By: Andrew Ng
URL: http://cnx.org/content/m45958/1.2/
Pages: 3-4
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 1"
By: Andrew Ng
URL: http://cnx.org/content/m45863/1.3/
Pages: 5-18
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 2"
By: Andrew Ng
URL: http://cnx.org/content/m45859/1.2/
Pages: 18-29
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 3"
By: Andrew Ng
URL: http://cnx.org/content/m45874/1.3/
Pages: 29-40
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 4"
By: Andrew Ng
URL: http://cnx.org/content/m45870/1.3/
Pages: 40-52
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 5"
By: Andrew Ng
URL: http://cnx.org/content/m45890/1.2/
Pages: 52-63
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 6"
By: Andrew Ng
URL: http://cnx.org/content/m45907/1.2/
Pages: 63-74
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 7"
By: Andrew Ng
URL: http://cnx.org/content/m45896/1.2/
Pages: 74-85
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 8"
By: Andrew Ng
URL: http://cnx.org/content/m45905/1.2/
Pages: 85-95
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 9"
By: Andrew Ng
URL: http://cnx.org/content/m45916/1.2/
Pages: 95-105
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 10"
By: Andrew Ng
URL: http://cnx.org/content/m45909/1.2/
Pages: 105-116
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 11"
By: Andrew Ng
URL: http://cnx.org/content/m45918/1.2/
Pages: 117-132
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 12"
By: Andrew Ng
URL: http://cnx.org/content/m45921/1.3/
Pages: 132-140
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 13"
By: Andrew Ng
URL: http://cnx.org/content/m45924/1.3/
Pages: 140-150
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 14"
By: Andrew Ng
URL: http://cnx.org/content/m45952/1.3/
Pages: 150-159
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 15"
By: Andrew Ng
URL: http://cnx.org/content/m45930/1.3/
Pages: 160-170
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 16"
By: Andrew Ng
URL: http://cnx.org/content/m45932/1.3/
Pages: 170-181
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 17"
By: Andrew Ng
URL: http://cnx.org/content/m45944/1.3/
Pages: 181-191
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 18"
By: Andrew Ng
URL: http://cnx.org/content/m45950/1.3/
Pages: 191-200
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 19"
By: Andrew Ng
URL: http://cnx.org/content/m45947/1.3/
Pages: 200-211
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 20"
By: Andrew Ng
URL: http://cnx.org/content/m45940/1.3/
Pages: 212-222
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 1 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45865/1.4/
Pages: 223-246
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 2 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45861/1.4/
Pages: 246-261
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 3 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45867/1.4/
Pages: 262-281
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 4 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45872/1.4/
Pages: 281-290
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 5 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45885/1.3/
Pages: 290-294
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 6 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45881/1.3/
Pages: 294-296
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 7a Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45898/1.3/
Pages: 296-298
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 7b Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45893/1.3/
Pages: 298-300
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 8 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45901/1.3/
Pages: 300-305
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 9 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45915/1.3/
Pages: 305-311
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 10 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45913/1.2/
Pages: 311-316
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 11 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45920/1.3/
Pages: 316-320
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Lecture 12 Course Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45954/1.4/
Pages: 320-331
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Review Notes"
By: Andrew Ng
URL: http://cnx.org/content/m45963/1.3/
Page: 333
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

Module: "Machine Learning Problem Sets and Solutions"
By: Andrew Ng
URL: http://cnx.org/content/m45957/1.2/
Page: 335
Copyright: Andrew Ng
License: http://creativecommons.org/licenses/by/3.0/

About Connexions

Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.