**Machine Learning:**

How I learned to stop worrying and love the algorithm

Hello everyone, and welcome to this afternoon’s talk on Machine Learning and artificial intelligence, which I have titled: “Machine Learning: How I Learned to stop worrying and love the algorithm.” It’s an old film and a cliché of a title, I know, but a) It actually sums up a lot of what I’m going to talk about today, and b) I’m horrible at coming up with titles. Also I love that film.

First up, a little background on me: I’m an undergrad at Yale who is studying music and mathematics, and I’ll be in my senior year this fall. I started getting into machine learning when I wrote a research paper on Noise, which is an actual class that an accredited institution will give you credit for taking. I got deeply interested in machine learning after I started dreaming up, with some of my professors, a system that could make connections in musical genres based on their content. Imagine if you could have a computer go through millions of hours of music from the last 80 years and tell me exactly when the Gibson Les Paul junior started making a comeback, just by *listening* to the music. Or if you could tell me how genres have shifted and merged, or how music is possibly categorized into VERY different genres than the ones that we put in Rolling Stone. Or, say, the influence of the songs of the White Album on the songwriting styles of contemporaries. Or what if you could simulate the sound of a violin with a synthesizer SO accurately by **listening** to thousands of examples of solo violin music? After doing a bit of research, I realized that ALL of these problems can be solved by machine learning. And we’ll get into how you can solve a couple of these problems in my talk in a bit. But I wanted to creep in my major because, as a side note, if you have any recommendations about good music you’ve been getting into in the past few weeks, PLEASE talk to me afterwards, because I’m a music addict and am suffering withdrawal symptoms as we speak.

Oh, one really quick thing also: **NO MATH WHATSOEVER** in this presentation. I’m not even going to try. While I’m a huge math nerd myself and would love nothing better than to derive the Law of Large Numbers for the next hour, I’m a big believer of not forcing your beliefs on other people, and I’m told by others that Math is a religion.

Before we get started, just to whet your appetite for the possibilities of machine learning, I’m going to show you some exciting news that’s developing in the world of machine learning right now, and it’s coincidentally very relevant to pretty much everyone in the room today, because it has to do with everyone’s favorite metric: predicting box office sales. From what I gather, everyone here is interested in box office sales— how to calculate them, how accurate they can be. But, of course, like any metric, there’s a large amount of skill required in predicting it correctly— and you need a lot of data to do it right. Here’s the question: how much more accurate would you like box office predictions to be? 75% accurate? 80%? That’s a pretty high level of accuracy, by anyone’s standards. What if I told you that Google just released a paper two weeks ago, detailing a machine learning process that they use to predict films— to within 94 percent accuracy, **four weeks before the release date**?

It’s a paper called “**Quantifying Movie Magic with Google Search**,” and it’s yet another example about how machine learning is changing the way that we can view and analyze data. How did they do this? They got a computer to **learn** what metrics predict movie sales, based on data from past movies. They do it using search data, as well as data from YouTube videos. They found that 48% of all moviegoers make their decision of what film to see on the day of their experience. They additionally found that 70% of the variation in box office performance “can be explained with movie related search volume seven days prior to the release date.”

How accurate does something like this get? Example: Google’s Movie Magic predicted that the Warner Brothers film, Magic Mike, would get just under 40 million at the box office. Warner Bros themselves predicted 15-20 million. The verdict? The movie made 39 million, which is pretty much exactly what Google predicted. For films with a far larger turnout, like “The Hunger Games,” Google predicted the opening box office sale to be $160 million, four weeks out. The result? $152 million in the first weekend.

Sounds exciting? Well, it turns out that it’s not just the search and film industry that can get excited about the possibilities of machine learning. There are literally thousands of benefits that machine learning has to offer for everyone— from the marketer trying to get an edge in his next campaign to the record company trying to catalogue all their old music. And, indeed, the number of academic fields heralding the dawn of machine learning is enormous. In fact, it’s so popular a catchall phrase nowadays that **dozens** of academic fields have published their own take on the concept— from computer science to game theory to music. I generated a quick word cloud that took about 200 of the most recent papers on machine learning and categorized by subject.

Now, it took me a long time to come up with this word cloud— I had to do the arduous process of going on Google Scholar, find all papers that have the word “Machine Learning,” in them, and then tally up the categories and sort by most used. Then, I had to properly format each category so that it could be read by Worldle, an online word-cloud generator, and copy the text in the number of times equal to the category count. The whole thing took me about an hour and a half of mindless clicking and categorizing: “Okay, so this one goes in the Game Theory category, this one goes in the “Computer Science” category. But my time was not completely a waste, because I’m making a point: Had I created a machine learning algorithm to categorize the articles instead, the whole process could have taken about 10 seconds.

So machine learning can save you time. It can save you a LOT of time categorizing, clustering… you name it. Machine learning has one other amazing property: It can also teach scientists and researchers a lot about how we **humans** make the decisions that we do. Think about it this way: If we can get a computer to simulate exactly the process that we humans make when we learn— so well that we can’t tell them apart— we know a LOT about what we need to have computers interact with us like human beings, not computers. And the more that we can get computers to think like humans, the more the user can understand the thought process of a computer and relate to it on a deeper level. Think **Siri,** or **Google Now.** If Google Now doesn’t remember that you asked it five seconds ago about the same flight from Boston to Los Angeles, you feel like you’re interacting with a search engine, not an A.I. But the fine engineers at Google have made Now *learn* your preferences, like whether you’re going on a trip next week, or what sports team you like. It makes for a much more human, personalized experience.

But here’s the problem: a lot of people talk about machine learning like it’s the second coming of computers and will solve all of our problems. Others look at it more like it’ll lead to the apocalypse, and Terminator II will just happen overnight because we’re teaching computers to do all these things. But one thing is clear: forget the debate, forget the controversy. Not many people actually know what machine learning is actually *doing*. Take my word-cloud example. If you solved this problem with machine learning, how would you go about it? Would the computer all the types of articles in some database somewhere? Would someone have to program in the categories beforehand? How does the computer “know” that one article is in a category or another? What happens if it gets it wrong? And, most importantly: How do we even **know** when the computer is making right guesses and wrong guesses? Do we have to go in to the data ourselves and just check? And, if so, wouldn’t it have been faster just to categorize the articles ourselves in the first place?

Now, before we can talk about how to apply machine learning, we have to understand what it’s doing on a deeper level, and what we’re capable of getting it to do. And to do that, I’m going to play a game. Actually, I’m not going to play just one game, but 10,000 games of my favorite board game: **checkers.**

In checkers, the rules are simple: capture all of your opponent’s pieces before they capture all of yours to win the game. There are only two types of pieces— the basic piece, men, which can move up to two directions in a turn— and kings, which can move in up to four directions per turn. The problem with checkers is— at least for me— even though I love the game and the rules are simple, nobody ever wants to play with me. So I end up playing many games of one-sided checkers with myself, and that’s no fun at all. I would love it if someone could play checkers with me, but at the moment nobody is interested. So what if I got a computer to play the game along with me?

Now how would you go about such a monumental task? Maybe your first inclination would be to just tell the computer to make a move— any move that is legal— and then prefer to make moves that get rid of my pieces or make kings out of pawns. But this method doesn’t take into account any strategy in the actual game. It’s just concerned with what’s going on with the current move on the board. It has no knowledge of moves say 5, 6 moves ahead of it, or potentially stupid moves that it might have done in the past.

Your second thought might be: “What if we just hard-coded in the **best** possible move in a given situation, and then have the checkers program play that one?” Now in theory, this sounds like a great decision. Get the best checkers player in the world, Grandmaster Gerry, to make a move given any situation, and then write that move into the computer, have it call a database of moves, and BAM: instant checkers genius. Now though it sounds like you’re well on your way to making a capable checkers AI, there are some serious problems. First: your program is only going to be as good a player as the person who programmed it. This is of course no problem if you have the God of checkers on hand with you, but usually that’s not the case. And, though mathematicians generally make good chess/checkers players, I’m definitely the exception. Also, this program never gets any better at playing checkers— it’s perpetually stuck playing the same game that Grandmaster Gerry played when you wrote the program. Finally, it’s incredibly costly, in terms of time AND money. You have to get Grandmaster Gerry to find the best move for EVERY position on the board, which is 2.3 sextillion, for those of you counting at home— not exactly something you can get done on a weekend.

So what do we do? Well it turns out that Arthur Samuel was faced with this same question in 1952, and he came up with an ingenious solution to the problem. Here’s what he did: first, he programmed the basic rules of the game into the computer— what pieces it can move in a turn, how to calculate the points it had, and when the game finishes. He then “played” the computer in chess, while the computer just spit out random moves— at this stage, every move is just as likely for it to win. When the computer inevitably lost, Samuel played it again. But this time, the computer responded differently: because it lost last time, every move that it played last game was assigned a *slightly* less chance of happening again. Samuel played the computer again and again and again— hundreds of times, in fact. Each time, the computer would assign a slightly worse probability to moves that it remembered were in losing games. And, on the off chance that it won the game, it would assign a slightly greater probability to all the moves it did on that turn.

After a few thousand games, the computer had a pretty good idea of moves that would cause it to lose. Because it remembered how each past game turned out for every position on the board, it could accurately predict what moves gave it a better chance of winning. Believe it or not, this system was so effective that, after several thousand games of checkers, Samuel started losing pretty consistently to the computer. The computer now had a sophisticated database of probabilities for every given move, and Samuel just couldn’t keep up. It had beaten Samuel at its own game. [[1]](#footnote-1)

I use this illustration for two reasons: First, Samuel was actually the first to coin the term “machine learning.” He defined it as this: “*A new field of study that gives computers the ability to learn without being explicitly programmed.”* But even without this context, the game of checkers demonstrates a few key points about machine learning. I call them the fundamentals. Number one: machine learning (in most cases) deals solely in **distributions of probability.** The computer never had absolute certainty about whether a move was good or bad, but it could adjust the chance of making good or bad moves with time. This doesn’t ever give us the certainty that we would have gotten if we hard-coded in the *best* moves into the system, but if you run it enough times, it’ll get pretty darn close. This brings me to my second point: machine learning requires **iterationto perfect.** Samuel had to play the game thousands of times before he had a program that could even win a few games, let alone one that good match his terrible skills. I presume that Samuel had a LOT of time on his hands to play several thousand games of checkers against the computer, but you better believe it was less time-consuming than somehow calculating the best move in any situation through game theory. But to do that, we needed the computer to evaluate itself— whether it won or lost— and feed that data back into the algorithm. This brings me to my third point: machine learning **(almost always)** requires **evaluation of existing data**. Another way of saying this is: machine learning requires *learning.* It’s something you might take for granted, but it’s an important step. We needed a way for the A.I. to store, access, and update the relative probabilities for every decision— so, we needed a database of all moves possible. We had to use the answers we got from previous iterations— whether it won or lost— to our advantage in future plays. This is called **supervised learning**. We gave the computer the answers— whether it won or lost the game— and used that to predict what to do the next time it plays the game.

Luckily, this algorithm isn’t limited to checkers. In fact: ANY problem in which you have these three things— acceptable margin of error, time, and data— can be solved with machine learning. So the bottom line: **if you’ve got a problem that we can solve to within an acceptable margin of error, one that you can spend computational time to perfect, and one that we know at least some data to start**, machine learning might be the right tool for you.

But let’s back up a bit. We might have a good approximation of what machine learning can do in practice, but then again, we were dealing with an example where someone ran a simulation 10,000 times. Is machine learning always that costly? Do you always need that much time, that much iteration, to perfect the algorithm? There are more questions that remain, too, like: what if our margin of error is really high? Do we just have to spend thousands of hours training the computer to perfect a model? How high does the margin of error have to be to be acceptable? And, most importantly, up until now we’ve been dealing with supervised data— we gave the computer some of the answers and asked it to perfect the model. What happens if we have fewer answers to give it?

I can give you an idea of that problem with another example. Another big place where machine learning is being used is in the world of **computer vision**. We’re dealing with actual computers, using the input from cameras to draw conclusions about objects in the real world. The problem is an extremely complicated one, but there are several steps that computer vision experts almost always include. Computers have to process images, understand patterns, and then compare those patterns to other that its either currently seen or has seen already. (Coincidentally, the Wikipedia page for computer vision has a word-cloud very similar to mine on its front page. I think mine looks better, though.)

Say we want a computer to recognize and tag pictures of cats on the internet— a noble task. If we show a computer 100 different pictures of cats and ask it to deduce something, the problem has to be broken into many parts. It sounds complicated, but many of the parts mimic features that our brains do when we compare objects. First, the computer has to take the vast amount of data from cat pictures and **compress** and preprocess it into a format that it can work with.Then, it starts **extracting very small features** from the images— maybe it realizes that hair is a common feature to all the pictures, or two eyes, or a nose. Much of the data it gathers is irrelevant to our cat-quest, but the computer has to, at some point, figure out which features are **most common** to cats, and start **classifying** those features. Then, at some point the computer has to **decide** whether what features it’s seeing all add up to a cat.

It sounds like a complicated process, but it’s one we perform every day— and at a very fast rate. We break down an object into its different components, and compare them to other objects that we’ve determined to be cats. “Does it have pointy ears and a coy smile?” “Yes, and I think every other cat I’ve seen has those properties.” “Does it have black fur?” “Yes, but I think that cats can have either black fur or white fur or brown fur.” You see how it works? It’s actually a process that mimics the way that humans do things.

**Here’s the problem**. In order to train a computer what a cat is, we have to give it plenty of examples of what a cat looks like. Just like the checkers example, we have to give it hundreds, if not thousands of examples for it to start drawing conclusions. But what if we don’t have the time to give it so many examples? After all, a set of 100 pictures of cats is not going to represent the huge amount of variance of cat pictures on the internet. What if all 100 of the pictures that we give the computer are of orange cats, or are simply pictures of cats’ faces? The algorithm will do well with cats that have exactly the same face, but HORRIBLY against cats with the slightest variation. In machine learning, this is called **overfitting**, and it’s a serious problem. It means that, when we trained our algorithm, it had so few examples that it basically just memorized the attributes of each example, and figured that those were all of the features that it would ever encounter.

So what do we do? **Do we define each feature** of a cat for the computer carefully ourselves, give it examples, and label each a part, each perspective, a cat? Give it a much larger set— say, 1,000,000 cats that we defined, and have it go and make further conclusions? WRONG— that’s not the machine learning approach. This process is not **extensible**— because we had to spend time defining what features to look for, we can’t easily apply this algorithm to look for, say, dogs or rabbits. We’d have to re-do the definition process all over again, and that’s no good. And if our training set is too big, we’ll run into other, bigger problems— the computer will start finding connections in images that have almost NO connection, but only because we asked it to look harder. This problem is called **data snooping**, and can be devastating to the conclusions that you make.

It turns out that **neither of these solutions** are the proper way to solve the problem. More precisely, **all of these solutions** are proper ways to solve the problem— we just don’t know which one is the best way to solve it. The key is in the **algorithm**. It turns out that there are dozens of way that we could define what a cat is and what it isn’t. We could define it by the amount of hair, number of eyes/ears, etc. At SOME POINT, our definition of a cat has to rule out other animals as well. There has to be a line in the sand drawn as to what is classified as a cat, and what we throw out as some other animal. That’s the job of our algorithm. The **algorithm** is just our method of approaching our final hypothesis— that is, what we define as a cat, and what we leave out. Our hypothesis comes from a set of all possible hypotheses, and the algorithm gets to *learn* which hypothesis is the best.

Take a look at these graph, for example. You can see that I placed are 100 red shapes and 100 blue shapes, and they all tend to gravitate toward the same space. There’s a pattern to the data, but I’m not telling you the answers. One question is: how do we create the best line to separate these two categories? A better question is: how do we pick a line that will best separate these two categories ***when we add more blue and red balls****?* We could pick a line to divide the two. Or we could put a circle around each of the categories, and widen it when we get a farther out result. Or we could calculate the distance to the center of each cluster. All of these are possible solutions— but, if we want to generalize, we need to make sure that those solutions are possible by our algorithm.

It’s actually a Catch-22: If we make our algorithm **too simple**, then we can only possibly have simple solutions to our problem. Think the line, for example. If we try to include more complicated solutions to our problem, the algorithm has to be more and more complicated. And, if **Occam’s Razor** has taught us anything, the simpler of two solutions should always be the one that you choose, if you get the same results.

So all this is to say that machine learning is not EXACTLY automatic. In fact, it requires a great deal of planning, depending on what you want to accomplish with your program. If you have a lot of example data, or we have a LOT of time to find a good model, your algorithm can take more luxuries and approach a more complicated model. If we don’t have one of those things, we need to be a lot more careful picking the algorithm to use. This is why literally hundreds of specific algorithms have been made for specific applications for machine learning. Different priorities are needed, and often, there is ***no inherent property*** in the problem as to which is the best priority.

Take this example: a **fingerprint scanner**. This is a classic machine learning example: it has you scan your finger a number of times to get a sense of the unique pattern on each of your fingers. Then, when you go to scan your finger again, the algorithm matches your new scan with its database of old scans and finds the best fit. Now imagine that this was a fingerprint scanner in a supermarket, and you had to scan your finger to get a discount on what you’re buying. What’s the level of acceptable error? For a discount, a false positive isn’t a very big deal: if the program identifies you as someone else who doesn’t get a discount, it’s not the end of the world if you end up giving them a discount in the end. Conversely, a false negative is definitely a little worse. If you’re a customer who *does* get a discount, you’re not going to be too happy if the program can never correctly read your fingerprint. More than likely, you’re going to storm out of there if it happens too many times, and the supermarket loses business. So in this example, having false positives is okay, but false negatives are not generally a good idea. Contrast this with a fingerprint scanner at, say, the NSA. Now the fault tolerance for a false positive is ZERO. You don’t want anyone sneaking into your secret labs where you spy on everyone in the world. Your false negative tolerance is definitely higher, though. It’s okay if the program rejects you a few times if you’re actually an NSA employee. After all, you get paid to work there— you can take the time to swipe in a couple extra times.

What’s the fault tolerance of our checkers model, for example? In that, **the margin of error changes gradually over time**— we’re improving a model, so we’re okay if the margin of error for the first couple thousand is pretty high. We made the algorithm to account for that: we adjusted each possible move by a very small percentage, and that got us toward our final solution over a slow amount of time. It’s okay if it gets a few moves “wrong:” after all, it’s only a game, and it’d be no fun if the computer won every time. With our cat example, however, the margin of error is definitely higher. We don’t want to be categorizing things that aren’t cats as cats… or worse, making up a whole model that picks the wrong attributes for choosing a cat. So our algorithm might actually choose from a number of different, lesser algorithms and pick the most efficient one, and we’ll use a large sample size of existing material to ensure that we’re not overfitting the data. We know ahead of time that we’re dealing with large objects that are made up of a hierarchy of parts— the cat’s ears, head, etc. So we’ll use a model that is generalized enough— something that can cluster parts together, and then cluster those clusters together to make more clusters of clusters of clusters.

And so, I’ll add one more caveat: I’ve already mentioned that you should use machine learning if you’re okay with some margin of error in your predictions. And I’ll add one more: in general, the more generalized you want your data to be, and the fewer training examples you give it, the higher your margin of error is going to be. In fact, it gets to be MUCH higher… as you increase the number of things to measure for and find connections with in your algorithm, the margin of error increases **exponentially.** So don’t go thinking that you can just build one system that will solve all your problems perfectly. It almost always has to be a compromise between giving the computer information, and having a lower precision and/or recall.

And, In case you were wondering about that cat example, it’s been researched heavily by none other than… **Google’s engineers**. Led by Jeff Dean (who else?), the program began looking for patterns in YouTube videos, and come up with high-level objects that could represent those patterns. How many videos did it have to watch to come up with “cat” as a concept? Google doesn’t say, but at one point it was using **16,000 distributed computers** to work on the problem. One amazing thing is that it never defined what a cat was to the computer. The machine just learned from seeing recurring patterns again and again. Pretty amazing stuff.

Now, I think we have a pretty good idea of the fundamentals of machine learning. We’ve covered the fact that machine learning is good when you can solve problems to **within a degree of error, when you have time to perfect the algorithm, and when you have data to support it.** We’ve also looked at some of the tradeoffs you have to make when you’re analyzing large amounts of data, and how you have to take into account a lot about the data set, how general you want the solution to be, and how much error you’re willing to accept. Now, with this in mind, I think it’s time to return to the algorithm that we started with: **Google’s Movie Magic algorithm**. Now, I don’t have access to the algorithm itself. I can only go by the methodology that they outline in their paper. But I think I have a pretty good idea of how they solved the problem, and, as a matter of fact, it uses variations of nearly all the examples of machine learning we’ve seen so far.

You see, the problem has two distinct parts: on the one hand, Google needs to use their petabytes of data on everything to predict whether someone is going to see a movie or not in the opening weekend. So they need to take a LOT of the data that they have and place it into specific categories. “I’m very likely to see the film,” all the way down to “I wouldn’t see the film if you paid me.” For each of these searches, they have to determine what the probability is that the searcher belongs in one category or another. So they solve it in the same way as we would categorize internet cats. In machine learning terms, we would categorize this as a **classification problem.** Google has to classify the likelihood of a user seeing the film, and their willingness to buy tickets, by classifying searches like “Iron Man 3 tickets” separate from “Iron Man 3 trailer.” For one, the searcher is MUCH more likely to see the film than the other, so they tally people accordingly.

Google also has to classify a lot of things about the film, such as its genre, actors in the film, how big the budget is, and what studio is producing it. Google’s paper says that for each film, they gather 30 parameters to do with the film, and they group films that have similar attributes together by **classification**— just like we did with the cat example. They found that simply classifying types of searches and types of films could predict sales at the box office to within 92% within a week of release, and they did this through classification. They needed a small margin of error, so they made sure that they could train with LARGE sets of data… 10’s of millions, in fact. After all, they’re Google!

But this problem has another part to it, and that’s the part that makes this problem **appear more like a game**. The game is “Guess the Box Office Results!” Every time a movie is about to come out, Google plays the game with a machine that has dozens of models in its arsenal. At every week before the box office release, it can predict film sales with a different percentage of each model: Sometimes it uses only Adwords data, and sometimes it uses a mix of data from Google+ and movie trailers. And, just like Arthur Samuel’s checkers program, it self-evaluates the chance of “winning” for every weekend in time, using combinations of different metrics. After categorizing movies into different types, it learned from the failures of each model for every category. Google’s machine only had to iterate this process 92 times, and suddenly they had the solution! They found that a model using trailer search terms and franchise status won out every time! It was accurate to within 94%, 4 weeks in advance from the date of the film!

The best part of this approach: Their model can get better, and they don’t even have to tell it what to do. If, two years in the future, the best way to predict box office sales is some other metric (say… views from Google Glass), the model will adjust itself automatically! Thanks to machine learning, it’s extensible, general, **and** highly accurate, because it uses a massive amount of existing data.

So there you have it. Machine learning saves the day, yet again. Its flexibility allows it to be used in literally any of the hundreds of disciplines I mentioned before. It can play games against its opponents and win— even if its opponents are the creators of the program itself. It can discover patterns in data that you never thought to look for, and make connections that are far too advanced or subtle for us to notice. And, what’s more, it can do all of this at an incredibly fast rate— a rate that, with the exponential increase of processing power on distributed systems, is only about to get much, much faster. All you have to have is a data set, an expectation of error, and time to perfect it, and you’re well on your way. Thanks again for inviting me to give this presentation, and I hope it was helpful!

1. Alpha-Beta pruning and a rote function. [↑](#footnote-ref-1)