NaturalLanguageProcessing-Lecture02

Instructor (Christopher Manning):Okay. Hi, everyone, and welcome back to the second class of CS224N. I'm sorry about last time. That was a bit of a disaster with the lack of projection. I'm sure it was hard for everyone to try and work out what was happening on the different slides. It was especially hard for me because I guess with the small print down there, it was actually almost impossible for me to read what was on my own slides. It was reminding me of old Ronald Reagan press conferences when he couldn't read what was on the teleprompter and he started trailing off and couldn't see what was going on. So I guess that's so long ago now that some of you weren't even born in those days. Okay. So today, what I want to do is actually get intensively into a topic matter, which is to talk about how we go about what we call "smoothing." And in NLP what smoothing essentially means is we have this problem in NLP that languages have a lot of words. How many words languages have is something that's kind of sometimes debated in popular media before it became the age of web indexes drooling over how many pages they had. In an earlier age, dictionaries used to drool over how many words they covered, and therefore, how many words are in English. I mean, in some sense the answer is there isn't an answer, because the number of words is infinite because you can make bigger words by doing various morphological changes to words so that you can take some verb and then add things onto it, especially with sorta Latin words. So that's when you get these words like antidisestablishmentarianism where you're sticking lots of x's onto words. And you can kinda do that productively. Okay, but nevertheless, there are a ton of words, right, so big dictionaries list a couple hundred thousand words. And so that means we have a lot of stuff that we want to try and estimate probabilities for in a probabilistic NLP space. And that's very hard to do and doing it right is very important for NLP system.

And so doing that is essentially the topic of the first assignment, and I'm gonna talk about that today and then get back to machine translation systems, which is more of the sorta exciting application stuff next time. Just a couple other quick administrative things. So the first section will be this Friday. So it's at the same time, 11:00 to 11:50 and it's going to be on Skilling 193 on campus or in SATM it's going to be on Channel E2. So that section is essentially gonna be going through concretely examples of different ways of smoothing language models and the kind of things that you should be doing for the first programming assignment. Speaking of which, after this class is a great time to start work on the first programming assignment. And then for readings, there are lots of things that you can read about language models and smoothing. The key main reference is chapter four of the Jurafsky and Martin reader. And sorry I haven't updated quite a lot of the syllabus last time. There are actually the right readings that are sitting there now. So definitely read that. In particular for the reader, if you order a copy you immediately get online access to chapters four and 25. So you're right ready for the readings for this week. But then on other things that you can read, so in my book, Foundations of Statistical NLP, chapter six of that is also good. And then there are these particular papers and articles by Stanley Chen and Josh Goodman, which are kinda a state of the art on all the ways to do language modeling, so you probably don't actually want to read them. But if you want actually to look at a reference for more details on something, that’s the first place that you should turn.

Okay, so what are we gonna be about here? I mean, essentially I see what our topic is here is building models of language that we should know about what languages are like. And so questions that you might want to know about what a language is like are, well, you know, English speakers, what things do they say? And then that’s just the language internal question. A follow-on question is then what do the things they have to say, how do they connect to the world or commands and questions about the world? We’re gong to talk about that later for semantics, but for right now, I’m gonna talk only about what kinds of things do people say. And one answer to that is, well that’s the kind of thing that grammarians talk about and they work out structures of language and so on. And it’s turns out that a lot of that work hasn’t really addressed what kinds of things people say in the sense that’s useful for statistical NLP and in the sense that we’re gonna to develop with probabilistic language models. And there’s some little examples to motivate that. Here’s a sentence: “In addition to this, she insisted that women were regarded as a different existence from men unfairly.” This actually comes from a long ago student paper that someone submitted to a course when I was teaching a freshman seminar. And I think – I think it’s fair to say that in that sentence there’s nothing ungrammatical about it. The grammar of that sentence is perfect. And yet, I think it’s also obvious that this sentence just wasn’t written by a native speaker of English, no offense to the non-native speakers. That it’s just got this kind of unusual wording and feel and the ideas don’t somehow seem to make quite the right sense together. It seems like clearly not the way a native English speaker would say things. And effectively that’s the kind of – not for picking out non-native speakers, but in working out what the structure of the language is – that’s exactly the sense that we will want to use.

We’ll want to know, okay, speakers of English, what word sequences do they use? How do they put ideas together? What are the common ways of expressing certain ideas? If we have those kinds of tools, then we can build predictive models of natural language processing. And how will we do that? By and large, the way we’ll do that is by grabbing lots of online texts and then we’ll be able to do counting effectively. We will count stuff in online texts and we’ll know how languages are used. And so that’s essentially our tool today is first count stuff, and then we try and do a little bit of clever stuff to get it to make it work better. So the thing – when we have big amounts of human language stuff that we do things over, they get referred to as corpora. So corpora is just Latin for body. So it’s bodies of texts. So the singular of corpora is a corpus. Now those people who didn’t do Latin in high school, which is probably nearly all of you, that somehow picked up a little in a math class or something have this very bad tendency to pluralize corpus as corpi, but that is not correct, because it turns out as well as second-declension nouns that end in “us,” like radius, there are fourth-declension nouns that end in “us,” and the plural of corpus is corpora. Okay. Well let me motivate this idea of the language models that we’ll use. And the way I’ll motivate it is from speech recognition, which is precisely where the idea of probabilistic language models was first introduced and used extremely successfully. So in the next five slides, I’m going to give a very, very baby and quick rundown of what people do during speech recognition systems. This is the only thing I’m going to say about speech in this entire course. If you really want to know how speech recognizors work, you should do Dan Jurafsky's course, CS 224S that's taught in the winter.

Okay. So I speak and there's vibrations that cause air to vibrate coming out of my mouth and there's these vibrations just like the vibrations that a loudspeaker makes. And so we end up with a speech signal, and you've probably seen something like this in – on your computer when it's showing you a sound signal. And essentially all you can see if you look at it like that is the amplitude, the loudness of the speech. But if you blow it up and you blow it up and you blow it up, you start seeing that human speech looks like this. It doesn't look like a sine wave, exactly, but you see that there are these very regular patterns that are repeating. So there's one regular pattern here. And actually, sorry, where this is pointing is on the line, right, so this is the "o" sound. And after that we change into "a" sound, and you can see that there's another regular repeating pattern. And so those complex repeating patterns are what we use to understand speech. And so the way we do it is we take these 25-millisecond samples, we do electrical engineering stuff, signal processing and fast Fourier transforms and things like that, and at the end of the – and you take these samples every 25 milliseconds – we take a 25-millisecond bit of speech overlapped every ten milliseconds. You need 25 milliseconds to effectively be able to do the Fourier transforms successfully over the kind of hertz rate that people speak at. And so at the output of this, you get these vectors of real numbers every ten milliseconds. And I'm happier with vectors of real numbers and ignoring the signal processing. Okay. And so I'm – this is a view of a spectrogram speech people actually use to further derive form. This gives the general idea of what happens in speech. So when you look at the – at which frequencies there's energy when people are speaking, that what you see is these dark bands that are called formance. So essentially the idea is that our vocal cords are vibrating and so there's a base rate of those vibrations, and then that sets up harmonics as multiples of that base rate of vibrations. And then it goes out through our mouth and some of those harmonics are damped by our mouth and some of them are accentuated and so it's the shape of the mouth that then determines where you get these excentered harmonics, which are then called the formance. And they kinda can tell us about valves. This is – which is then kinda high-energy frequenters and so that gives a distinctively different pattern.

Okay. Enough about speech. How do we do speech recognition? Well your speech recognition essentially become something that worked around 1980 when people started applying probabilistic models to speech recognition. And the way this was done is by setting up things as a generative model of speech. And so the idea was we assume people have in their heads some things they want to say, a sequence of words. And then what they do in some clever way, they encoded it as an audio signal and it comes out into space. We hear, with some further degradation as it travels, that audio signal. And what we'd like to do is reconstruct the most likely sequence of words, given that audio signal, which means we want to find the arg max over w of the probability of w given a. The sequence of words that is – over here – the sequence of words that is most likely given what we hear. And so the way we do that is by inverting things by base roll and saying okay, well, we can still say there's a probability of a sequence of words that people would say and then there's, for those words, there's a probability distribution over how they're pronounced, which would then be normalized. But since we only want to find the arg max, we can throw away the normalization and so this is our formula for speech. And in speech recognition in other part of IE, this gets called the noisy-channel model. And NLP people also commonly still refer to the noisy-channel model. For the noisy-channel model is nothing more or less than base roll, which I assume everyone has seen previously. Okay. And so then we end up with here the probability of words being said a certain way, the acoustic model, and the probability of sequence of words, the language model, which is what we'll focus on today.

But first a question. I mean, what we wanted to do was find the most likely sequence of words given the speech signal that we heard. And what I've done is changed that into we have to find out two things. Why is that progress? It could seem like this is negative progress, because if you started off with one thing that you have to estimate and I say, oh look, I've made it better. Now there are two things we have to estimate. That doesn't exactly sound like a deal yet. So why is that a helpful way to think about things? Yeah? Student:

[Inaudible].

Instructor (Christopher Manning):Yeah. So I think it's more the second one. Well, I mean, it's not exactly that they're both simpler, right? Because this is no simpler than that, right?

Student:[Inaudible].

Instructor (Christopher Manning):Oh, at any rate, these are things that's easier to see how we could estimate. So it seems like we could estimate the probability of word sequences. We collect a lot of things that people write or say and we work out the probability of word sequences. That seems doable. It seems quite clear how we could work out the probability distribution over how the words – the way words are pronounced. We get people to say words and we record their pronunciations and that gives us a distribution. So these are much more estimable, whereas how do we work out that? Well I guess we could try, but it seems a lot harder. Okay, well, the kinda interesting thing is if you start thinking about this model of the noisy-channel model, it turns out that there are lots of things that you can apply the noisy-channel model to. And one of those things is, indeed, machine translation. And actually, interestingly and very famously, the person who kicked off machine translation research in the United States was this guy, Warren Weaver. And Warren Weaver was a prominent government science and policy official after the end of the Second World War. And so he – and he was essentially the one that convinced people that machine translation was going to be essential and drove a great deal of funding in the direction of machine translation, as I'll come back to again next time. Well the early work of machines translations didn't exactly go the way Weaver thought it would and wasn't very successful, but what his original thinking was that well, in the Second World War computers had been very successful for the purposes of code breaking. And so the way he was clearly thinking is foreign languages, they're essentially just another code. And so why can't we decode them? And so he suggested the following. Also, knowing nothing official about, but having guessed and inferred about the powerful new mechanized new methods in cryptography, methods which, I believe, succeed, even when one does not know what language is being coded. One naturally wonders if the problem of translation could conceivably be treated as a problem of cryptography. When I look at an article in Russian, I say this is really written in English, but it has been coded in strange symbols. I will now proceed to decode. So essentially has in mind this noisy-channel concept that let's assume it started off in English, it's been coded in this funny way, and we can decode back to the source. And so statistical machine translation started when people developed that idea and paid it out. So suppose we want to translate from French to English.

The original work in statistical machine translation was from French to English and somehow from that the nomenclature has stuck and the equations that you see in machine translation papers always have E and F where you want to translate from F to E. But you can just think of them as any two language. Suppose you want to translate from French to English, though. Our model is that French speakers really think in English. They have in their heads sentences of English. French might not like that part, but let's just assume that. So they have a sentence in English, but they code it in this funny code, and words come out of their mouths in French. So what we do when we want to understand what they're saying is we start with this French and we decode. We say what's the most likely underlying English sentence that would have been rendered as this French sentence? So we want to find the arg max of E given F. And we split it up exactly the same way by base roll and we have a probability over sequences of English words and a probability of ways of things to be translated. Note, crucially, that this is exactly the same. It's exactly the same concept of the language model, which is just over English. Depending on whether we're doing speech recognition or translation, we'll need something different here for the channel model that the language model is unchanged. Okay. And it turns out that once you get thinking this way there are just lots and lots of problems that you can think of as noisy-channel processes. So take just one of these examples, Google spelling correction. How do they do it? They do it as a noisy-channel model that they're assuming underlyingly everyone can spell perfectly, but somehow they make boo-boos as it comes out of their fingers and that their job is to reconstruct the underlying perfect spelling.

Okay. So we want to build probabilistic language models that assign scores to sentences. So I want to be able to take a sentence like "I saw a van," and give it a probability. And the reason I want to do that is so I can make comparative judgments, that it's not actually exactly what this number is that's important. I want to know whether things are more or less likely. And I will use this when I'm translating to decide what's a good translation into English? It should be one that's the kind of thing that English speakers would say. So, crucially, this isn't kinda some kinda strict notion of grammatics. It's the kind of things people would say. So "artichokes intimates zippers." That's an okay sentence grammatically. It's got a subject, a verb, and an object. They agree, etc. But we want our language model to say it's really, really unlikely that that's the good translation of this sentence. Okay. Well one way we could do that would be to take a bunch of sentences and the ones that occur, we give them a count and we divide through by the number of sentences and say that's a probability distribution over sentences. Now why doesn't that work? Yeah?

Student:

[Inaudible].

Instructor (Christopher Manning):We assume that everything has been said before. Yeah, so the only sentences that we give non-zero probability to are ones that we heard. And that's just not true. Language is a creative, generative process and people invent sentences that they've never heard before all the time. And so what we want to do is generalize, and essentially there are two tools that we'll use to generalize. The idea of backoffs, so sentences will be generated by little bits, which we can recombine together in different ways. And then we want to allow the possibility of people saying things that we just haven't heard bits of before, and so that'll come out as the idea of discounting, allowing for things that we haven't seen before. Okay. So the first step in working out the probability of sentences is to apply the chain rule. So if I want to work out the probability of word one, word two, word three, I can without loss of generalities. Say that's the probability of word one times the probability of word two given word one, times the probability of word three given words one and two. No assumptions. Just by itself, though, that doesn't help me, because working out the probability of word three given word one and two is no easier than working out the probability of word one, word two, word three as a sequence, right? That's how you calculate conditional probabilities. So the crucial tool that we use, and it's where the n-gram model idea comes from, is that when we want to estimate a sentence, we don't fully use the chain rule, but we make a Markov assumption and assume that only local context is necessary.

So that we can take not the product of word 17 given words one, word two, word three, word four, word five through word 16, but we'll decide that the probability of word 7 only depends on word six and five. So then we're just using a local context to predict, and that's a Markov assumptions. And all [inaudible] language models make a Markov assumption. Okay. So here is the most radical and simple Markov assumption that you can make. You can say each word is independent of all other words. So the probability of word one through n just equals the probability of the words. So you just keep on generating words independently, and in our vocabulary, we have one extra special symbol, which we call stop. And we give some probability of stopping. Why do we need a stop in there?

Student:[Inaudible].

Instructor (Christopher Manning):If we're going to have sentences, we have to stop sometime. If we just gave a probability of words and generated with the unigram model, we'd just have infinite. And for some purposes that kinda doesn't matter, but if we want to generate sentences, at some point we have to stop. So we effectively want to have some probability given the sentence is ending. Okay. And so this is probably too small to read, but these are the kind of things that you can generate with a unigram model. "That all limited the. Fifth and of futures the n incorporated a the inflation most all is quarter in his mass." Doesn't sound very like English. Okay. So unigram models are clearly hopeless. I mean, they're useful for some purposes, include code breaking. But we can't model what sentences are like with unigram models. But we can just start using a little bit of context. What if we use a bigram model? So a bigram model is a simple Markov chain model where each word is conditioned on the previous word. Note here a funny list of the way of counting. In the gram terminology, people number things on what are the size units that you have to count. So a bigram model, you're counting how often pairs of words occur, so that's the bi two of bigram. And a bigram model corresponds to a first-order Markov model, because that only counts the conditioning context and you're counting one conditioning thing. Okay. So if we go to a bigram model, things get a lot better already. "'Texaco rose one in this issue as pursuing growth in a boiler house,' said Mr. Gruyer." You know, doesn't make any sense. It's not correct. But it's sort of starts to almost sound like language. There are bits of phrases, right? "In this issue as pursuing growth." There are sorta bits of stretches of texts that sorta hang together. Okay. So, I mean, well, why stop at bigram models? Well people don't.

I mean, bigram models are essentially the first thing to use that's kinda sensible and does something useful. But people go to larger n-gram models. In my second note on classical education for today's lecture, the standard terminology most people use is to talk about bigrams, trigrams, and four-grams. Now, this is enough to send any classic scholar screaming from the room, and leave it to just the engineering kinds in this class, I guess. Yeah. So bigram is wrong, actually, because gram is a Greek root. You get pentagrams and tetragrams. Grams should be a Greek root. So, actually, we should say digram. And actually if you go back to the work of Claude Shannon, who invented the ideas of information theory and these ideas of using Markov models for modeling language, and if you actually look at his papers, because the state of high schools were so much better in those days, Shannon actually uses the word digram. But somehow that quickly got lost and so people use bigrams and trigrams and then, by the time people started doing higher-order context models, it just seemed too hard at that point and so people changed to English and talk about four-grams and five-grams. Whatever. I think it's kind of fun, in a way. Language use at work. Now, so these kind of very local language model, I mean, they're clearly not correct for English. I mean, English has this hierarchical structure that I'll come back to when we get to topics like parsing. And the classic example people show as to why we need more than this is you get these clear long-distance dependencies in English when you interpose stuff. So when you have something like a relative clause, "the computer which I just put into the machine room on the fifth floor crashed," that "crashed" is the verb of which "computer" is the subject. And the reason why the verb "crashed" should be likely is because computer occurred very early in the sentence.

And that's not something that we'd be able to predict with a bigram, trigram, four-gram, five-gram, six-gram, seven-gram model. It's way back in the context. So ultimately we'd like to use more context. And people have pursued that, so people have used tree-structured language models. But, in general, they're still just well beyond the state of the art that anyone uses to do anything practical and useful. And it turns out that despite the fact that they're sorta wrong, these n-gram language models just work incredibly successfully most of the time, and that’s largely because stuff like that doesn’t happen a lot. Most of the time the dependencies are very local and you can capture most of what’s going on on local structure with things like trigrams. Okay. So how do we now – now that we have our bigram model, say, how do we go about calculating probabilities? Well this is easier in essence. What we do is we take a large amount of text and we just count how often pairs of words occur, and then we divide through by how often the first word of the pair occurred in total and it gives us estimates. So here is our corpus: “I am Sam. Sam I am. I do not like green eggs and ham.” And commonly we use a start of sentence symbol.

This is just to make the equation a little bit – equation code a little bit cleaner, because now we can condition the start of sentence on a unique symbol. And then we have this is our stop symbol. Okay. So at beginning of sentence, there are three beginnings of sentences and two of the sentences started with “I.” So the probability of “I” at the beginning of sentence is two-thirds. That makes sense. So this thing is called the maximum likelihood estimate, and in most of statistics, maximum likelihood estimates are good things. And they’re good things because they model the data really, really well. A maximum likelihood estimate is the thing that makes the data on which we built the model as likely as possible. If you want, you can go home and do a little bit of calculus and prove that that’s the case. If you don’t want to do a little bit of calculus, it turns out that for these counting generative – generative models over discreet events – the maximum likelihood estimate is always that you just take relative frequency. No harder than that. Okay. So that sounds easy and good. Yeah?

Student:[Inaudible].

Instructor (Christopher Manning):Oh, sorry. This is the count of the number – sorry, this is sliding a bit that says the notation. It’s the count of the notation of times that something occurred. So for here, this is the count of start of sentence “I” is two over the count of just start of sentence, which is three, and this is writing it up in a complex way that means that it would work for n-grams of any order. Just ignore that. Okay. Normally maximum likelihood estimates are good things. Why are we going to have problems with this for natural language processing? Yeah?

Student:[Inaudible].

Instructor (Christopher Manning):Okay, so a lot of possible sentences, and we’d need a lot of data to count.

Student:[Inaudible].

Instructor (Christopher Manning):It only gives probability to things it’s seen before. So what a maximum likelihood estimate does is if you’ve never seen a word somewhere, the probability estimate you give it is zero. All right? That’s precisely this notion here. Makes the observed data as likely as possible. If I said, oh, there’s a chance of a sentence starting with the word “green,” even though I didn’t see it in my corpus, the effect of that would be to make the observed data less likely. But it turns out that that's precisely what we want to do. What we'd like to do is have a model that says even though we've never seen it, it's possible to start a sentence with "green eggs." And so we're going to do something other than that. I'll just give you one more example of that. This is a project that Dan Jurafsky worked on many – long, long ago, when he was a grad student. It was the speech project, the Berkeley restaurant project, which you also find in the textbook, where they were collecting sentences of people saying things about Berkeley restaurants. "Can you tell me about any good Cantonese restaurants close by?" And so here a bigram counts from this corpus. It was only a small collected corpus, but you can see the kinds of things people say. They say, "I want," and they say "I want to eat," and sometimes they even say, "Eat Chinese." And they say "Chinese food" a bit. And there are other things that happen just occasionally. Occasionally you see "I – I." That's because when people speak they repeat themselves and stutter in that way, and they say "I – I want to eat." Okay. But notice that most pairs were never seen at all. Are there pairs in there that are pairs that you'd expect to see sometimes?

Student:[Inaudible].

Instructor (Christopher Manning):What's that?

Student:[Inaudible].

Instructor (Christopher Manning):"Spend lunch," yeah. We spend lunch playing video games, or something like that. Yeah, that's a perfectly possible one. Any others?

Student:[Inaudible].

Instructor (Christopher Manning):What's that?

Student:[Inaudible].

Instructor (Christopher Manning):"Food to." We gave food to the homeless person, yeah. Right, so there are lots of pairs here that are perfectly plausible, and indeed, sorta seem more correct than the repeated "I – I" that was observed in the corpus. And so if we turn these things into unigram – turn these things into maximum likelihood estimates, the problem is all of those things are deemed impossible and that that is bad. And I'd just like to go and spend a minute of saying, well, why is this bad, which leads up into evaluation. So how do we know what's a good language model, anyway? So we want our language model to give a high probability to good sentences and a low probability to bad sentences, where good sentences means ones that somebody might say or write in the future, and bad sentences are ones we don't expect someone to say or write in the future. And so the way we do that is we start off with our set of data, which we call the training set. And we build a model. Now the problem is we can't test our model on the training set, because if we test our model on the training set, the best model to build is a maximum likelihood estimate that says exactly the sentences we saw are exactly the ones we're going to hear again. There will never be any new sentences. In fact, we should also, then, forget about all of this bigram stuff. We should just say look at these complete sentences. These are the only ones you can utter again. So that's not how it works. So to work out how good our model is, what we need is some different data we can test our model on. So we need a separate – a separate data that's called the test data, which has to be disjoint from the training set, and it's preferably something we've never looked at. So it's real hidden test data. And then if we run our model on that and see if it's – how it's scores, if it scores well in some sense, then we know it's good data – I mean, it's a good model. Okay. Well, how can we work out if it scores well?

There are two fundamental ways in which you can work out if a model scores well. One is doing an extrinsic task-based evaluation. So if we actually want to see whether our language model helps our speech recognizer, we can plug it into a speech recognizer and see if our speech recognition rate improves. And so for speech recognition, the most common evaluation is this thing called "word error rate." And so the way you do that is you have the correct answer, which is what a human says the person said. You have the recognizer output, and then you count how many insertions there are, how many substitutions there are, how many deletions there are. You run the kind of edit distance dynamic programming to find the best match. Then you take the total of those and divide through by the number of words in the correct answer. Note crucially, although everyone quotes these as percentages, that means that a word error rate can actually be over 100 percent, because if you're far enough wrong you can have more edits going on than there are words in the correct sentence. Okay. In principle this is the best thing to do to do these kind of extrinsic evaluations. But there are two problems. One is that it takes a long time to train up speech recognizers, so you kinda tweak your language model and you maybe have to wait two weeks before you find out whether it's better or worse than the last one. So that kinda kills off your experimental cycle. And secondly, it means that how well you do isn't just a feature of your language model. It's also tied into with how good your speech recognizer is and how the two interact together and various things. So it'd be nice to have somewhat evaluation that just looked at the language model. And so the standard thing that people do there is play what's called sometimes the Shannon game, because this is something Claude Shannon suggested.

And what we want to do is predict the next word. So "when I order pizza, I wipe off the – " What word might come after that? Sauce? Grease? Hopefully not gasoline. Yeah. So Shannon actually literally got human beings to do experiments to see human beings' ability to predict the next word in this way. And human beings can, in a lot of cases, do it surprisingly well. And so, using all of the context and world knowledge available to them, there's a fairly low level of uncertainly. Not zero, of course, because if I start saying, "yesterday I bought a – " Well, you can guess the kinds of things I might be likely to buy, but you really don't know whether I bought a monitor or a car or a whatever. Okay. So we formalize that in this measure of what is the entropy of a piece of actual text, according to some probability model. So we've made a probability language model, then we take a text, which is a sequence of words, and we work out the entropy, the uncertainty, measured in bits of those words according to the model. And so the way we do that is we work out the probability of each word, which can be conditioned on the preceding words according to the model, and then we kinda sum up the logs of those probabilities. And that will give us the cross entropy of text according to our model. And what we want is for that cross entropy to be as low as possible, because low entropy means we're pretty certain about what words comes next. And so that's the entropy measure, which is sorta the correct one to use. In practice, though, most of the time people in the NLP field and speech don't actually use entropy, they use something that was called perplexity. Perplexity is you simply take the exponential of the entropy. So you if your entropy is five bits, meaning kinda on average you're – it's like there are 32 equiprobable choices for what comes next. You exponentiate that and say perplexity is 32.

I joke there that the reason that perplexity is better than entropy is it gives you more impressive numbers to report to your grant sponsors, because it sounds a lot better if you can say you got perplexity to go down to from 250 to 175, rather than saying it went from eight bits to 7.9 bits. To tell just – no, I shouldn't tell you more stories. I should get on, since I got behind last time. Okay. Okay, so we've got a trainer model on a text corpus and build a model. So what actually does a lot of text look like? So just to say this very quickly, everyone read Tom Sawyer in school, right? It's a book. It's not very long. Really we need a lot more text than that. But I can look at the words in Tom Sawyer and if I do, the commonest word is "the." The second most common word is "and," then "a," then "to," then "of," than "was," then "is." It's all these little words that get called function words, the kinda grammatical words. The first real content word we come to is Tom. And I guess that one shows up with the frequency it does because of the obvious reason as to what book I used. What's more important to our purposes is these are the really common words that occur hundreds of times in the book. Actually, of the words that are used in Tom Sawyer, there are about 8,000 different words that are used in Tom Sawyer, and of those 8,000 words, almost exactly half of them get used precisely once. And so if you think of this from a point of view of statistical estimation, the task that we want to do is kinda almost ridiculous. We wanna be working out the properties of different words, and yet half the words that we saw, we saw only once. That sounds bad. But to some extent, try and do the best job we can about that is exactly what people do when building language models. Then even when you go up, there's another 1,300 words we saw twice, 600 we saw three times.

All of these words we barely saw at all, but somehow we want to predict their behavior. Well one answer to that, of course, is to remember that Tom Sawyer's not a very thick book, and so, of course, we'll want to have more text. And we do. The more texts we have the better. But nevertheless, there's this huge problem in working with language is just this problem of sparsity. And that happens at all levels. So even at the level of words, all of the times you keep on seeing new words that you haven't seen before, either real ones, like someone says some word like "synaptitude," and you haven't seen it before. Or there are various classes of other things, like IP addresses that turn up that you haven't seen before, as well. But it's not, in practice, it's not even so much the problem of totally new words. Because really what we like to do is work – make bigram models and trigram models. And so for building those, the question is, have we seen this two-word sequence before, or have we seen this three-word sequence before? And that's where things really start to get bad. So in this corpus, as we go up to a million words, well, there are still new words regularly appearing, but their rate is relatively low. The problem is that for bigrams, actually about a quarter of the bigrams that we keep on seeing and ones that we've never seen before. And if you go to trigrams, it's about half or more of the trigrams at this level are ones that we've never seen before. Four-grams, barely would have seen any of them before. So that the vast majority of the time we just haven't seen the high-order context that we want. Okay. This fact about the sparsity of natural language goes under the much-heralded Zipf's law.

It says if there's this weird and somewhat crazy guy who discovered the fact about language, it started with language, that if you put words and ordered them by their frequency, which I started to do with "the," "a," "an," "to," "of," etc., and then you took the product of the word's frequency and its rank order in that listing, that that product becomes – is roughly constant, which is equivalent to saying that if you take log rank against log frequency, you approximately get a straight line. Here it's being shown for the Brown corpus. So here are all the words that we heard once, a ton of them. This is a million words now. All the words that occurred twice heading all the way up to this is the word "the." Okay. So – so Zipf was completely convinced he was onto the most amazing discovery in nature from having discovered Zipf's law, and he set about, then, showing that Zipf's law distributions occur absolutely everywhere, and so in every possible domain that was available to him, he started finding Zipf's laws. So the kind of Zipf's laws he discovered were things like wars and the number of people killed in them, number of rail lines that connect into different cities, all sort of things of that sort. This enthusiasm hasn't waned, and people are still doing this on the worldwide web, just about everything you can measure on the worldwide web has a Zipf's law distribution. So number of links to a page, lengths of pages, lots of things. But it's not actually as profound an effect as that. Because when you start thinking about it, there's something a little bit shonky here, because the rank of a work is intimately tied into its frequency, because we rank them according to frequency. But at any rate, the real effect that Zipf's law has is that in this picture here, although these are rare words, in aggregate, there's still a lot of probability mass down there.

So in standard statistics, when you start off with normal distributions and work forward, the whole beauty of normal distributions is because of that one over the things squared bit inside the exponent, that that means that the normal distribution tails off so quickly that there's almost no probability mass out in the tails, whereas for language you get what are called heavy-tailed distributions. And anything that's a Zipf's law curve is a heavy-tail distribution. When in aggregate there's still lots of probability mass that's out here. Okay. So somehow we want to be, when we build our probability models, say, lots of stuff that we haven't seen before we're going to see in the future. And so what we need to do about that, and I get to the main theme of the class eventually, is smoothing. So the idea here is we're in some context "denied the." And we've counted on a fairly large corpus and we've seen some words that can follow. "Denied the allegations," "denied the reports," "denied the claims," "denied the request." We've seen seven words coming after "denied the." And that's quite typical, right? Because probably if we collect ten million words of data, we'll have seen "denied the" about seven times. These things aren't that common in language. Okay. And so here's our maximum likelihood estimate and we give no space to other words, but “denied the attack,” that’s worth something that you’d want to have be possible. So what we’re going to do is smooth in the sense of erosion. We’re going to take some of the probability mass away from the these things that we – we have seen, and give it to the guys that we haven’t seen. Okay. And I’ll just skip that. So what I want to do is, to the extend I have time, then talk about five ways that we can do smoothing.

Okay, and here’s where I eventually introduce the notation of what does the c mean. Okay. But we – assume that we’ve got a training text, which is of some number of words as tokens. We can work out counts of unigrams and bigrams. We have a vocabulary size, and I’ll come back to the vocabulary right at the end, where we assume that we have some list of words, which can just be all of the words that we found in our training corpus. And then later, in just a minute, we’re also going to have this big nk, which is the kind of things I was showing with Tom Sawyer. So n1 is how many different words were seen only once in the corpus. n2 is how many different words were seen twice in the corpus, etc. Let’s go to the concrete example. Okay. So the very famous philosopher and statistician – I forget his first name – Simon-Pierre, I think. Simon-Pierre Laplace came up with famously about 157 years ago Laplace’s law of succession. And Laplace’s law of succession was that the way to allow for the possibility of unseen events occurring in the future is what you do is you have, well, you have something that you’re wanting to predict, which has some set of outcomes – set of outcomes is our vocabulary of all words. And what you do is you take the actual number of times that something has occurred and then you add one to all of those counts. And so then you’d have the total amount of data plus the size of the space of outcomes and this gives you smoothing. So the idea here is if you want to predict whether the sun will rise tomorrow, you have a binary outcomes rises tomorrow or not, and you’re 20 years old. That means you’ve seen, whatever, 7,000 – no, no, 70,000 days. Whatever. You have – your chances are one over 7,000 of the sun not rising tomorrow. And your chances are 6,9999 over 7,000 of the sun rising tomorrow. Okay. People have done things like CS 228 uniform [inaudible] prior, is what this is.

Okay. Well does that work well? I’ll skip the fine print there, though it’s good to think about in your own time, and let’s just try it and see what happens on this Berkeley restaurant corpus. So here what I’ve done is added one to all the observed counts and then I can divide this through and have gotten these probability estimates. And an obvious reaction is to say, “Jeez, I don’t know. These might be all right. How can I tell?” And an easy way to think about that is to turn these probability estimates back into expectations, which are counts. So that is to say, well, if these are my probability estimates and I generated a corpus the size of the Berkeley restaurant corpus, how often would I expect to have seen each word pair? And if you do that and get reconstituted counts, they look like this. And then if we flip back between this one and that one, we should start to get an idea is this good or not. To take just one example, after “want” in the actual training data you saw various things, but you almost always saw “wants to,” which kinda makes sense in this context. It’s you want to do something. So the estimate for “want to” should be really high. I mean, based on this is obviously only a partial display of the complete bigram matrix, but based on the data we can see it seems like the probability of “to” after “want” should be .9-something. Okay, what happens when we’ve done Laplace smoothing? Well, the probability of “to” following “want” has turned into

.26. Bad news. That doesn't seem to correspond to our observed data. And, indeed, if we look at the expected counts now, it seems like we're out by a factor of about three. It used to be 690 or something, and it's now 238. So in our predicting, the "to" will follow "want" about one-third as often as it did in the observed data.

That seems like a really bad thing to have done, because that doesn't accord at all with what we saw in the training data. Yeah?

Student:[Inaudible].

Instructor (Christopher Manning):So these counts were – so in an earlier slide there were the actual empirical counts as to how often pairs of words actually happen. Here I've added – here I've just added one to the empirical counts so I can do Laplace smoothing, and then I divide through by to normalize to get probabilities. Then for the next slide I say, okay, overall in the Berkeley restaurant corpus I saw whatever it was, 65,000 words. So what I'm going to do is take 65,000 times "want to" – no, sorry, I'm saying that wrong. Sorry, that was wrong. Rewind. In the Berkeley restaurant corpus I saw "want" – is that sorta 1,000 times – I saw the word "want" 1,000 times. So the number of times I predict "to" to follow "want" would be about 260, because – and so I, therefore, work out, based on my model, what is the predicted number of times I'd expect to see a word pair in a corpus of the same size. Does that make sense? Okay. So it seems like this has worked out kinda badly. And so the question is, why did this work out okay for Laplace when he was predicting whether the sun was likely to rise and why does it work out kinda badly for me? Anyone have an idea on that, if we go back to the equation?

Student:[Inaudible].

Instructor (Christopher Manning):So sort of yes, that's the answer, but there's a slightly crisper way of thinking about it and saying it.

Student:[Inaudible].

Instructor (Christopher Manning):Yeah. He had only two outcomes and I've got thousands and thousands of them. So Laplace's law of succession works pretty sensibly if the space of possible outcomes is small. It doesn't have to be two. You can have ten. It works fine, because typically you'll have a reasonable amount of data. You might have a couple hundred data points and if you're adding on ten here it works kinda fine. But all the time when we're estimating these n-gram models for a particular history, so this is what words follow "house," or what words follow "desire," or something like that. That particular history we might have seen 100 times or we might have seen it only five times. So this number is small, where this number here might be 20,000, it might be a million, depending on the size of the vocabulary used. So this number becomes orders of magnitude bigger than this number, and Laplace's law of succession works crazily because what effectively it does is it gives almost all the probability mass to unseen things and barely any to seen things. I mean, in particular, and this is an idea that other smoothing methods pick up, is if you saw something very commonly that just using the maximum likelihood estimate should almost be right, you don't actually want to muck with it much. If your maximum likelihood estimate is that "to" follows "want" 90 percent of the time, well, if we – if we are going to give some probability mass to previously unseen things, we have to lower it down a little, because it's essential that we make it a probability distribution that adds up to one. So we have to make things a fraction smaller, but we kinda don't want to move it below .89 conceptually. If we've seen "want to" happening hundreds of times, that's one of these rare things that we've actually been able to estimate accurately and well. And so we actually do know what the chance of "to" following "want" is. And so anything that's cutting that estimate by a factor of three is obviously crazily wrong. So a little hack you can try is say, well, why not instead of adding one, like Laplace did, let me add a smaller number. I can add a quarter, a tenth. And actually that does make things a bit better. But it doesn't – nevertheless, any of these kind of add methods of smoothing actually don't work very well for NLP applications, because of this problem of the big vocabulary and sparseness and are basically not used.

Student:[Inaudible]. Have people tried just moving the edge of [inaudible]?

Instructor (Christopher Manning):I think the answer to that is no. I mean, that could be an interesting thing to think about and try and do a project on. I mean, yeah, I mean, you're completely right that in NLPs that are done, that by and large, the only thing people ever use for anything is point estimates. And yeah, I mean, clearly I agree that it's a good observation there that some of these estimates have fairly tight confidence intervals and some of them could be all over the map, and there's an idea of how to exploit that. But I guess at the end of the day of your probability model, you do have to choose a particular estimate. Okay. So well, another intuition – this add smoothing is doing this funny thing of adding counts, but a way to think about that and what it's doing is that we have a count of how often an event occurred, and we turn it into a probability just by normalizing. So what we want to do is reduce that observed count and pretend the count was a little bit smaller, so we shave some mass off the seen things, which leaves some count mass for the unseen things. And so the question then is how much should we shave off? So if we saw something r times in our c samples, the maximum likelihood estimate is r over c. And what we want to do is shave a bit of mass and be left with r*, which is a little bit less than r, and then we use r* divided by c as our estimate. So maybe empirically what we might want to do is actually shave half a count off everything. So if we saw something 860 times out of 1,000 things in the context after "want," the maximum likelihood estimate is 860 out of 1,000. Maybe you want to shave of a half and say it's 859.5 counts divided by 1,000, which would be .85 -- .8595, which would have the desired effect that we just shave off a little bit. And so one simple way to work out a smoothing system is just to work out such a discount empirically. So this is sometimes referred to as Jelinek-Mercer smoothing, because they prominently first did it in speech work. And so the way we do this is that we get a hold of a second set of data, which is neither our training set or out test set, so it's a development set. And we just simply empirically ask ourselves this question. We say, let's look at the things that occurred five times in our training data. How often, on average, do they occur in our development data? And maybe the answer is that they, on average, occur 4.3 times.

Some of them occur seven times, some of them occur two, some of them didn’t occur at all. But on average they’ll occur around, but a little bit less than, five times. So the average might be 4.3. And so we say, okay, well what we could do is subtract .7 from the observed counts of everything and use that shaved off count mass to give to unknown things. And that kinda method actually works very successfully. And, indeed, it’s the basis of some of the most used smoothing methods. So this is this idea of discounting that I actually saw something n times, but I assume – I pretend for my model that I saw it a little bit less than n times. And so every time I do that, I’ve shaved off some probability mass that I can reuse to give to unseen things. And then the question is how do I give it to unseen things? And the dominant idea there is n-gram modeling is we decide how to give it to unseen things by lowering the order of the context. So that if I have a trigram model and I’m trying to predict “Hillary saw,” what words came after that? Well I’ll have seen a few things, and so those ones I can estimate and shave a little. But a lot of things I won’t have seen in the context of “Hillary saw.” So how might I work out what’s likely to occur there? Well, I can back up and say, well let me just consider words that occurred just after the word “saw.” Anything that occurred after the word – I’d seen after the word “saw” was presumably kinda likely after the words “Hillary saw,” even though I haven’t seen it. And conceptually then there are two ways that that can be done, which are referred to as backoff and interpolation. So in backoff, you use a higher-order estimate if you have it, and otherwise you do an estimate using a lower-order Markov context. Whereas interpolation, you effectively mix together estimates of all orders at all times. And so let me just quickly show you interpolation and then I’ll get back later to a backoff example. So here’s a very simple way to do a bigram model with interpolation.

You choose a number lambda between zero and one. You do a linear interpolation between the bigram model and the unigram model. And exactly the same thing can be extended to trigram models and you just use two lambda. If you want to do the simplest possible thing that isn’t silly, the simplest possible thing that isn’t silly is to use a trigram model for when you’ve got sorta one to ten million or more words of data. And do a linear interpolation of different order models, where the models, if they kinda backoff then these individual models can actually just be maximum likelihood estimate models, because you’re using the lower-order models to give you smoothing. And mix them together with sensible weights and such a model actually works decently. It’s not a bad thing to do. Much better than thinking that you could do something like Laplace’s smoothing. Nevertheless, though, we can do considerably better than that. And one of the ways that we can do better than that is rather than having lambda be a constant, we can make lambda be itself a little function that depends on the preceding context. And so the general idea here is if you’ve seen – if you’ve seen the preceding context commonly, you should be giving more weight to the higher-order models, because they’ll be reasonably useful and accurate. Whereas, if you’ve seen the preceding context very rarely, anything that you’ve – any probability estimate from a higher-order context will be extremely pointy and not a very good estimate, and you’re better making more use of lower-order estimates. Okay. So that this is this idea of having an extra set of held-out data. So very commonly in NLP we have at least three sets of data, training data, held-out data, and then our test data. And we use the held-out data to set some extra parameters. So if we have this question of, well, what’s a good way to set lambda?

Well one way is just to guess and say I'll put a weight of .6 on the bigram model and .4 on the unigram model. And that won't be crazily bad. But a better way to do things is to say, well, I can test on some held-out data and find the value of lambda that optimizes performance. And similarly, we use the held-out data for the Jelinek-Mercer smoothing to assign how big a discount to put in. Okay. I think the time that's available right now, I'll do Good-Turing smoothing and then I'll stop and then I'll say a little bit more about a couple of other things on Wednesday. Okay, but the next – Good-Turing smoothing is these days not the most common, but it's sorta the second most common method that's actually used for smoothing estimates. And intuition between Good-Turing smoothing is related to Jelinek-Mercer smoothing. And the way Good-Turing smoothing works is as follows. This is Josh Goodman's intuition or story to tell, because Turing's smoothing. Let's suppose you're out fishing and you've caught some fish. You've caught ten carp, three perch, two white fish, one trout, one salmon, and one eel. So in total you've caught 18 fish, if you count the eel as a fish. And then how – if you're thinking about what's your next catch is going to be, how likely is it that you'll catch something that you've never seen before? And the answer proposed in Good-Turing smoothing is that your chance is three-eighteenths. And the way to think about that is that you're thinking about each catch going through in sequence. So I start off with no fish in my boat and I caught – start catching fish. Well how often am I caught something – and I'm doing a bad job of explaining this. I'll erase that bit and restart. The idea of how you come up with this estimate is that you're looking at the things that you only caught once, and those things that you caught only once are the basis of there being three of those and your chances of seeing something new again.

Well what does that mean for the things that you did see? Well, for the one trout, your chances of seeing a trout again can't possibly be one-eighteenth, because you're saving some probability mass for unseen things. So it has to be less than one-eighteenth. Okay. So I'll try and do it right this time for explaining Good-Turing Reweighting. Okay, so one way of doing things is with Jelinek-Mercer hold-out data set. The disadvantage of using a hold-out data set is you kinda need to have data that you're using for other purposes, rather than for estimating your model. And so that reduces the amount of data you've got for estimating your model, which, on balance, makes your model less good. So somehow we'd like to be able to use all our data for building the model we can and make it as good as possible. So Good-Turing smoothing adopts the idea of well, we can essentially simulate the process of doing Jelinek-Mercer smoothing. In particular, we can do a limit case of it, which is corresponding to leave-one-out validation. So we do the following experiment. Here's my whole corpus and let me pretend that one token wasn't included in it. And let me use the remaining c minus one tokens and try and guess about the one token I haven't seen. Well is that token going to be a token that I've never seen before? Well sometimes it will be, if the word that I took out was a word that occurred precisely once in my training corpus, then the word that I have kept out, it will be a brand new word that I've never seen before. And so what I'm going to do is I'm going to do this little thought experiment, in turn taking each work in my training corpus out and calling it the hidden token that I haven't seen. So if I repeat that experiment c times for the c bits of data, the number of times that the word I now see will be a work that I've never seen before will be precisely the number of tokens – the number of word types that occurred once in the data. Because each time one of those word types occurred once in the data that they'll now be a new word. And so that's the basis of this idea that you get to this number of three-eighteenths. And so then for one more idea, well, what happens for the words that were – what happens for the words that are actually in the training data k + 1 times?

Well when I do my thought experiment, one instance of them will be hidden, and they'll be k instances of them left in my thought experiment data. So for data – when I've actually – in my thought experiment when I've seen a word k times, the weight predicted is to look at the words that occurred k + 1 times in the actual data, and to use them as the basis of prediction. And so that's what Good-Turing smoothing does. And so you're working out that if you've seen something k times in your thought experiment, the number of times that happens is the number of words that occurred k + 1 times in your total data times k + 1, because you see them that many times, divided through by a normalization constant. And so that gives you a basis for estimating those higher-order words. Okay. I better stop there for today and maybe I'll say a few sentences more about that next time. But you guys can also think about and read about Good-Turing smoothing as well as everything else.

[End of Audio]

Duration: 76 minutes