N-grams

Professor Eric Atwell: Today, I'm going to be talking about n-gram language modeling. This will be based on chapter 3 of the Speech and Language Processing textbook by Dan Jurafsky and James Martin. OK, so let's start off by saying, I'm going to introduce n-grams and to try to explain what these are as a way of modeling a sequence of words or other things. Then you have to figure out how to estimate the probabilities of each n-gram.

Having got a language model with n-gram probability estimates, how do you evaluate that? And one way is to measure perplexity of the language model against a corpus. And we'll look at generalization, because you're only counting up things in your training set. What happens if there are no examples of a word in your training set? You get zeros.

Well, one way to deal with that is what's called smoothing. And we'll look at a very simple add-one smoothing or Laplace smoothing. Another thing you can do is backoff or combining n-gram models by interpolation. And these are handy tools for very large web scale language models.

OK, let's start off with what's the point of probabilistic language models. You want to assign a probability to a sentence, a score of some sort. And a probabilities are nice, because they're in the range 0 to 1. So mathematicians like these, compared to just scores of some sort of number. Why would you want to do that? I mean, language is about understanding the language, not working out some number value for the language.

Well, there are various applications. In machine translation, for example, if you want to have a French sentence translated into English, should you say high winds tonight or large winds tonight. High and large are both big things. Well, it turns out, in English, high winds as a collocation is more likely than large winds. So high winds sounds better than English. And we capture this by capturing the probability of high winds as being greater than the probability of large winds.

Another thing is in spelling correction. Microsoft Word wants to underline words which are not correct. And if they're not in the dictionary, then they're clearly not correct. But sometimes words are in the dictionary, just less probable. We want to say that 15 minutes from has more likely, more probability than 15 minuets from. And this is to do with minuets being less likely than minutes. But also after a number or in the phrase, 15 something from minuets has a lower probability than minutes.

Similarly, in a speech recognition, if you get a sound like I saw a van, well that's possibly the word I followed by the word saw followed by the word a followed by van. Or it could be eyes, awe, of, an. It's another way of segmenting the speech signal. But you want to know overall the probability

of the sequence I saw a van is more likely than the probability of a sequence eyes awe of an, even though both of those are made up of regular English words.

And the same sort of rationale applies to summarization question answering and so on. Typically, you want to compare two or more possible word sequences to work out which one is the most likely, the most probable, and probably the most meaningful. So we want to compute the probability of a sentence or more generally of a sequence of words.

So we have word one, word two, word three and so on. And we want to work out the probability of the whole sequence. Similarly, if we can work out the probability of a sequence, then we can use this to compute the probability of the next word. So given word one, word two, word three, word four, what is the probability of the next-- of w5 being the next word?

So any model which computes either of these, either the probability of a complete sequence or the probability of a next word given the words up to this point, that's a language model. You might call this a grammar, although linguists don't like to call this a grammar. I think of grammars as being in things involving nouns and verbs. And in English, you have to have a noun phrase followed by a verb phrase.

OK, but so that Language Model, or LM, is a standard thing in computational linguistics. So how do you go about computing the probability of a word sequence? Well, you might want to think for example, here's a nice sequence. Its water is so transparent that. Well, that's a sequence of words. And you might have a chain rule that it's water. Water is so-- so transparent-- transparent that. Those are chains, and the whole thing links together in some way.

So if you can compute the probability of the parts and then join them together. And in probability terms that's multiplying them together. And that should work.

So this is to do with conditional probabilities. And hopefully in some previous learning, possibly in the data science module, you will have come across this idea that you can have the probability of B given A, is essentially the probability of a sequence A followed by B divided by the probability of A. Or the probability of A followed B is the probability of A times the probability of B, given A.

There's different ways of reformulating this. In a word sequence, there's more than two words. And it gets a bit more complicated. So for example, the probability of A,B,C,D, in other words, the cat sat on, for example. A is the. B is cat. C is sat. And D is on. The probability of A,B,C,D, is the probability of A times the probability of B, given A, times the probability of C, given A and B, times the probability of D, given A,B,C, and so on.

That's a very general thing. So in general, we have this the probability of any sequence is the probability of the first word times the probability of a second word, given the first word, times the

probability of the third word, given the first two words, and so on. And you can use this to compute the probability of the words in a sentence.

So the probability of its water is so transparent is the probability of its, times the probability of water, given its times the probability of is, given its water, and so on. There's a slight problem here, that the given thing is everything you've seen so far. And this does work in principle. And remember these are probabilities in the range of 1 to 0 or 0 to 1. So to combine them, we multiply them together, and we have this nice sort of formula multiplied together, all the sequences or subsequences in the chain.

So we could just, in a large corpus, count up the frequencies of things to give us the probabilities. So for example, the probability of the, given its water is so transparent that. Well, we count in a large corpus, how many times do we come across, its water is so transparent that the and divide by the predecessor, its water is so transparent that. And that gives us the probability of the, given that what we've seen so far is, its water is so transparent that.

Of course, that doesn't actually work because you have to know-- to get these counts, you have to have a frequently recurring count of its water is so transparent that the, and the count of its water is so transparent that-- that doesn't happen. And in reality, you're not going to find long sequences occurring significantly. So that doesn't really work. That's never going to happen. We're never going to see enough data for estimating even the subsets.

So Andrei Markov, a Russian statistician, came up with a simplifying assumption about a, well over 100 years ago now. What maybe you could say, the probability of the, given that its water is so transparent that, is roughly-- just take the one word before. The probability of the, given that the previous word is that. So how many times do you count up that the, or possibly not that the but transparent that.

So you count up how many times do you get transparent that the? And how many times have you got transparent that? So Markov modeling also applies to letter sequences. In fact, if you go to Leeds University library, you'll find some of the works by Andrew Markov, where he counted up in a Russian newspaper individual letters and pairs of letters and sequences of three letters. That's because there's a smaller number of possible letters than there are possible words. It's actually, without computers, this is something you can realistically do by hand, whereas counting up word sequences is much harder.

OK, so we have the Markov assumption that the probability of a long sequence of words can be computed in terms of shorter subsequences. We approximate the component of each product. There's more details of this in the textbook. But let's say the simplest case, we just don't bother looking at the past at all. But we say the probability of first word, 2nd word, 3rd word, and so on it's just multiplying together, the probabilities of each of the individual words, regardless of the context.

And then we can use this to generate example sentences from the unigram model. Uni is the Greek word for one. So unigram means just look at the individual gram or in this case, the gram being the word, the individual word. And then if we'd have a random generator, which generates words at random but takes into account the probabilities of individual words, then you'll get random sentences like, fifth, an, of, futures, the, an, incorporated. And this is just taking words at random but favoring words, depending on their individual probability.

So you're more likely to have common words like, the and a and less likely to have less incommon words, like "inflation." But none of these are actually very good representations. So maybe you want to look at bigrams. Bi is the Greek word for two. And a bigram looks at pairs of words. So the probability of a word sequence is taken for each word, you favor it depending on what word came before.

And then we look at bigram sequences and then just use it to choose a word at random. And then the next word you choose, depending on the previous word. And then the next word you choose, depending on the previous word and so on. Then you get some slightly more English-like sentences. This would be a record November could be generated that way.

I'm going to extend this to trigrams or threes or 4-grams or 5-grams. After trigrams, you forget trying to use Greek words and just say 4-grams and 5-grams. And this works in principle, but there are always some exceptions. Language does have a long distance dependencies. For example, the computer, which I had just put into the machine room on the fifth floor crashed. Well, it's actually the computer crashed. But you've got-- so the generation of the word crash depends on the computer, more so than the other words in between. So that's a long distance dependency.

Luckily, in English, this is quite rare. So actually 3-gram, and 4-gram, and 5-gram models do work most of the time. So you can get away with simple models like that. OK, so that's an introduction to n-gram modeling.

Now, how do we go about estimating the probabilities of these n-grams? Remembering, you might say, you can count up in a large corpus, if you want to get the probability of bigram model, the probability of a word given the previous word, then you count up the pairs of those words. And you count up the individual word. And that gives you a probability estimate.

So for example, given three sentences, I am Sam, Sam I am, I do not like green eggs and ham. This is from a famous book for children. And notice as well as the words like I and am and Sam, we also have these sort of pseudowords begin a sentence and end a sentence. That's because the first word I is conditional on it being the beginning of a sentence. And Sam is followed by end of a sentence.

So now we can start to count up probabilities. So for example, the probability of I, given the start of a sentence-- well, there are three starts of a sentence. So we have three bigrams involving the start of sentence followed by I, start of a sentence, followed by Sam, and start of a sentence followed by I. So there are three cases where it starts a sentence. And all those two, are start of a sentence followed by I. So the bigram probability of I, given the start of a sentence is 2 divided by 3 or .67.

And you can think out further examples like this. If you're not sure about this, because you're not really happy about probabilities, then have a look a the textbook or read this over yourself. But you can figure out probabilities of any word given the previous word, as long as you include start of a sentence and end of a sentence as being possible words.

So for example, the probability of Sam being at the end of a sentence-- sorry, the probably of end of sentence, given that the previous word is Sam. Well, Sam occurs twice here, once at the end of a sentence and secondly at the beginning of a sentence. So given that you've got the word Sam, what's the likelihood that you've got the end of sentence, when it's one out of two or a half.

OK, that's a very simple example. We can count up more reasonably on a larger corpus. Well, let's for example, let's go to the Berkeley restaurant project corpus. So some people recorded typical sentences or conversations to do with booking places at restaurants. Can you tell me about any good Cantonese restaurants close by, or mid-priced Thai food is what I'm looking for.

OK, so there's some typical examples. Notice the text has been normalised. It's all been put to lowercase. But that's about all. So this is conversations. And from this, we can come out with raw bigram counts. You can count up pairs of words. There's only about less than 10,000 sentences. This is quite a small corpus compared to, say, the British National corpus. That's a 100 million words. This is much smaller.

And we can see, for example, I want occurred 827 times. Want to occurred 608 times. To eat occurred 686 times. So the obvious ones are quite common. Notice also you've got I, I occurred five times. So this is in natural conversations. And there are hesitations. So you can say, I, I, and that's probably someone who wasn't quite sure what to say. So I--I want to eat Chinese food, as an example.

Also I eat occurred 9 times. Want I occurred twice. I'm not quite sure what an example is, but it does happen. Furthermore, there's a lot of zeros. I too never occurred. I Chinese never occurred. I food never occurred. So all of these zeros-- in the table, you get frequencies of expected things, like I want. You get frequencies of unexpected things, like want I. And you get things which never occur, like want, want.

You also have to count up the individual, like I, and want, and two, and eat, so that you get a frequency. Let's go back to these frequencies-- I, I is vert-- that does occur five times, which is quite common, you might think, compared to some of the other things. But I is actually a very frequent word. So you want to have I, I divided by the frequency of I itself. So 5 divided by 2,533 is actually a much smaller number.

So if you normalise by the unigrams, then some of these hesitations and other words, which you shouldn't really want, go away. I want is still quite a high number. 0.33 is still quite good. We've normalised them, so their probabilities-- everything is in the range 0 to 1. You still got a lot of zeros. So in here, dividing by-- zero divided by something is still zero. So normalizing by unigrams doesn't really affect the zeros.

Now we want to estimate the probability of a sentence. Like start a sentence, I want English food, end of the sentence. That's the probability of I, given start of a sentence times the probability of want given I, times the probability of English given want, times the probability of food given English, times the probability of end of sentence given food. And from this table, you can calculate this probability. And it turns out to be 0.000031.

You can do this at home if you want. And this is based on knowledge of probability of various things. OK, so what we're trying to collect is the probability of a word, given the previous word. And so for example, it turns out that English is less popular than Chinese. So after want, you're more likely to want Chinese food than English food, for example.

Notice that these are probabilities. And then the range is 0 to 1. And if you combine probabilities, you multiply them together, which will give you an even smaller number, more closer to zero, a tinier fraction. This is a problem in computing terms, because you end up with underflow, if you trying to deal with very, very small fractions. So what you tend to do in computation, in the actual implementation, is convert everything into a logarithm.

Because a logarithm in logarithm space avoids underflow and also to combine logarithm, you add them together, which is faster than multiplying. So the actual implementation dealing with the logarithm as well as the probabilities. But conceptually, they're actually still logarithms-- still probabilities, or fractions in the range 0 to 1. And probably you won't actually implement this yourself in Python, but you'll use some sort of language modeling toolkit. Obviously within Python, there's lots of mathematics tools, but you can also go and use quite sophisticated toolkits for doing language modeling. And there's more examples of this in the textbook.

Now we've talked about counting up all this. Of course, you can count probabilities yourself from a corpus. Google have access to the whole of the web, so they have a huge corpus available to them. Back in 2006, a group of them got together and released the Google n-gram. So what they

did is they counted, they said they processed one million, million words, which is 1 billion or 1 trillion, depending on your British or American.

And they counted up all the 1,000 million five-word sequences that occurred at least 40 times. So that's 1,000 million or a billion, depending on whether you're British or American, that there are 1,000 million sequences of 5 words, which occur at least 40 times. And they got the frequency of these. There are also 13 million unique words, once you throw away all the words which appear less than 200 times.

So there's 30 million words which occur 200 times or more. Imagine that, 13 million English words occurring more than 200 times. So even misspellings may occur more than 200 times. You have to be careful about that.

So here's some examples. You can if you want to, download these yourself. But notice, it's a very big file. So beware. Serve as the incoming. Serve as the incubator. Serve as the independent. You will, if you look at this, you'll see some of these are actually quite rubbishy. They include misspellings of all sorts. But they happen to occur at least 20 times.

And you can go to ngramsgooglelabs.com and have a look at them. But I advise you not to unless you want to download a very big file. OK, so that's the end of that.

Now, let's have a look at n-grams and evaluation. How do you decide that a language model is good? Well, does our language model prefer good sentences to bad ones? Well, you can do this by looking at some of the sentences it generates. What you really want to do is, does it assign higher probability to real sentences-- are they really, rather than ones that are ungrammatical or bad sentences in some way.

Well, we train the parameters on a training set. This might be the whole of Google, Google n-grams, or it might be just your small training set, particularly if you're dealing with a language model for a particular application area. Let's say, I'm only dealing with text to do with text analytics. Then I don't want to use the Google n-grams, because that includes everything in English, not just text about text analytics.

So if a particular topic or domain, my research group at Leeds does a lot of research on the Quran and Hadith, so only want language models about Quran and Hadith. That's the religious text of Islam. We don't want to use Google n-grams.

We test the model's performance on data we haven't seen, which is a test set. So typically, we have to have a test set, which is different from the training set. This is basics in machine learning. You train a machine learning model on a training set and then see how good it is on a separate test set.

And we have to have some sort of evaluation metric to tell us how good the model is on the test set. OK, so best evaluation models for comparing A and B. So for example, we might want to train a model and use it on a task, like a spelling corrector or a speech recogniser or a machine translation system. So you run the task.

We have a language model A. And we have a language model B. And we get the accuracy on the task. How good is the spelling corrector, given language model A? How good is the spelling corrector, given language model B? And if it's better given language model A, then the spelling corrector is best.

So how many misspelled words are correct? Or how many words are translated correctly from machine translation system? You compare accuracy of the machine translation system for language model A, and for language model B. That seems to make sense. But unfortunately, it can be quite difficult. For one thing, we're saying, we first of all, train a language model. And then we try it out on machine translations of things, which is very time consuming.

What we actually want is the intrinsic evaluation. Extrinsic, x means outside. So we haven't got a language model. We then try it out on something else, like a machine translation system. Intrinsic, this is looking at only the model itself. Intrinsic is to do with, again, the Greek x means outside. In this case, it's Latin. X means outside, and in means inside. So I should explain, for those of you who learn English as a second language, a lot of words in English have roots or parts of words derived from Greek or Latin.

So in and x is from the Latin, just to confuse you. Sorry. OK, so intrinsically, we want to evaluate perplexity, which is how good the language model is, without trying it out on machine translation or spelling checking or anything else. It only gives us an idea of how good it is, which may work in pilot experiments. So really you should try it out on machine translation system, if you're building a machine translation system. But initially, when you're trying out different experiments for different language models, you can do it just intrinsically. It's helpful for initial experiments.

OK, so there we have this thing called the Shannon game. Shannon was an information theorist from over 50 years ago. He invented this game. It says how well can we predict the next word, take into account only a limited window. So for example, I always order pizza with cheese and. That's the window. How can we predict the next word. Or the 33rd president of the US was. Or I saw a. How do you predict the next word? Well, it depends on how many words before you can look at.

If you can't look at any words before, that's the unigram model. Then you just choose the most likely word overall. If you have a bigram model, and something or was something or a something, then you have better-- if you can look at cheese and. That's a trigram model-- cheese and

mushrooms, cheese and pepperoni, cheese and anchovies-- are much likely than cheese and fried rice.

OK, whereas US was isn't very likely to be mushrooms and so on. So unigrams are not very good for predicting. You have to look at some context, at least the word before and preferably two words before to get you a good prediction. So a better model of text is one which assigns a higher probability to the word that actually occurs, given some context before. And we know that intuitively, just looking at the word itself isn't enough. Looking at one word before is good. Looking at two words before typically is enough for most cases.

So the best language model is the one which predicts an unseen test set, given the highest probability. So perplexity is the inverse probability of the test set, in some sense, normalised by the number of words. And we apply this chain rule to give us for bigrams, a simplified model. So basically, the perplexity given-- we have a set of bigrams. In other words, the probability of a word, given the word before. If we take 1 over that, the inverse of that, and multiplied together for every possible bigram, and then take the n-th root of that, that gives us the perplexity.

If you want to find out more about perplexity, there's more description of this in the textbook. For data mining purposes, typically you don't want, unless you're a theoretician, a mathematician wanting to develop further more complicated complexity models, you're not going to need to know the exact maths behind this. This is the difference, I think, between if you like machine learning research and data mining research.

In machine learning research, you're developing new machine learning models. Therefore, we do have to understand the mathematics behind it. This is going to sound strange, but for data mining research, you don't need to understand the mathematics. Because actually you're using black box models. You're going to be using whatever Python has, whatever NLTK has for measuring perplexity-- and using the perplexity model to compare two models, to decide whether or not to train on the British National corpus or the Koran for your particular problem.

So this is heresy, but I don't really care about the mathematical models. If you do, read the textbook, and you'll find some more. But the point is that minimizing perplexity is the same as maximizing probability. And for example, if you take the Wall Street Journal corpus and train on 38 million words of Wall Street Journal, and have a test set of 1 and 1/2 million words of Wall Street Journal, then you can measure the perplexity of a unigram model, of a bigram model, of a trigram model and discover the unigram model has a higher perplexity. It's much worse at predicting.

The bigram model is a lot better. The trigram model is a little bit better. So in other words, for predicting the next word, bigram is much better than unigram. Trigram is a little bit better. And when you get to 4-gram, it tails off. Perplexity doesn't work so well. So in other words, perplexity as a measure tells you a trigram model typically is best.

OK, that's enough on that. Now, let's look at another problem, which is to do with zeros. The Shannon visualization method says, basically take a random bigram and generate its probability. And according to its probability, we can predict the next word. So I want or want to or to eat or eat Chinese or Chinese food and food end of sentence.

So we can predict the probability of I want to eat Chinese food as a sentence, by taking the bigram probabilities and multiplying them together. And that works fine, if there is in our training set at least one example of every bigram. If we string the words together, that gives us the probability of the overall string. We could do this for any corpus to generate sentences.

So for example, if we take the Shakespeare corpus, remember Shakespeare's got about nearly a million words, nearly a million tokens I should say, in the Shakespeare corpus. Therefore, we can count up probabilities of individual words and probability of pairs of words or bigrams and then use this to randomly generate Shakespearean style texts for different n-grams. So if we just take individual words, one grams in Shakespeare, then we have to him swallowed confess here both, which. Of save on trail for are ay device.

So this is basically generating at random, but where the random is dependent on the probability of individual words. It's not very good. Now if we try bigrams, then we get why dost stand forth thy canopy forsooth. He is this palpable hit, the King Henry. So we start to get more plausible things. For example, King Henry, the probability of Henry-- the probability of the next word, given King, Henry is very likely. Because King Henry, that pair is much more likely than King the or King hit or King palpable.

If we get to trigrams, it sounds even better. Fly and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. If we get to 4-grams, the probability of a word being dependent on the three words before, we get, King Henry, what I will go seek the traitor Gloucester. Exempt some of the watch. A great banquet served in. It cannot be but so. This is really Shakespearean.

Remember, Shakespeare as a corpus, there's less than a million words. And the vocabulary is about 30,000. So there's about 30,000 different words. So Shakespeare produced about 300,000 bigram types, out of-- remember that there's 30,000 words, so 30,000 squared is nearly 900 million possible bigrams. And of 900 million, only 300,000 actually occur. So most of the 99.96% or nearly all of the possible bigrams never occur. They have zero entries in the table.

Now this is a real problem. For quadirgrams, it's even worse. Almost all of the quadrigrams have a zero probability. The only quadrigrams which appear as one or more are ones that actually occur in Shakespeare. So if you start using the quadrigrams to generate random text, what's coming out will be virtually all things that occur only in Shakespeare and don't occur anywhere else.

Even if they're going to-- instead, try the Wall Street Journal corpus. The Wall Street Journal corpus is much bigger, therefore there's more examples of these things. And therefore if you start to generate bigram or trigram examples, they look a bit more plausible. For 3-grams, they also point to 99.6 billion dollars from 204063% of the rate. So you see, around numbers, you tend to get point and dollar and percent.

Unfortunately, these get mixed up in a 3-gram model. So the system-- a 3-gram model isn't clever enough to realise that if you've got dollars, then you shouldn't also have points and so on.

OK, you can guess the training set or author of the language model, because you can see these are not Shakespearean. The first one's clearly-- they also point to $96.6 billion dollars, blah, blah, blah. That's Wall Street Journal. This shall forbid it should be branded, if renown made it empty. That's clearly Shakespearean.

You are uniformly charming, cried he, with a smile of associating. And again, that's clearly Shakespearean. How is it that we can be so sure? Well it's because, when it comes to 3-grams, there are lots of 3-grams which only appear in Shakespeare. And there's lots of 3-grams which only appear in Wall Street Journal and not the other. So that 3-gram models are very differentiated for Wall Street Journal compared to Shakespeare.

And this is an example of overfitting. N-grams work very well for word prediction if a test corpus is from the same style as the training corpus. Once you get to trigrams or quadrigrams, then you very, very fit-- you fit the training corpus a lot. And that means it may not work for another corpus. In real life, this is not very sensible.

We need robust models which generalise beyond-- so it's no good having a very large Wall Street Journal corpus and then trying to use that for training on other stuff. And this is also a problem for Google's web corpus. They've used it for training on the web, which is huge. But their 5-grams are of web pages. And the English used on web pages is not necessarily the same sort of English that's used in Shakespeare or the Koran or the Bible or for other specialised training sets.

It's really a problem with zeros, things that never occur in the training set, but do occur in the test set. Zeros are a real problem. For example, if a training set contains denied the allegations, denied the reports, denied the claims, denied the request, but not denied the offer, then it says that denied the offer is not possible. Whereas intuitively, denied the offer sounds OK. It just didn't happen to occur in the training set.

So for trigrams, this is a real problem. If a test set contains denied the offer or denied the loan, and these are reasonable sentences, where they just didn't happen to occur in the training set. Then the trigram model will just say they have a zero probability. Bigrams with zero probability or

trigrams with zero probability, mean we assign zero to the test set. So we can't really compute perplexity, because perplexity involves dividing by a probability.

And if it's a low probability, we can do that. But if we divide by zero, we can't do that. So zeros are a real problem. So that's the problem with generalization. Trigrams extracted from one training corpus may not be appropriate for a different test corpus. And zeros are a particularly a headache for this.

So how can we get around this? Well, there's this thing called smoothing. And a very simple example of this is add-one smoothing or Laplace smoothing. Laplace decided this. So Dan Klein came up with this nice model. Where we have sparse statistics, so for example, denied the allegations occurs three times. Denied the report occurs twice. Denied the claims occurs once. And denied the request occurs once.

So we've-- even for allegations, we've only got three examples. And if you're a statistician, you know three examples is not significant. You need to have hundreds of examples to be sure-- to get hundreds of examples, you need billions of words of text. And that's just too difficult. The problem is in a million word corpus, if there's 20,000 words, then on average, each word occurs too rarely to get enough examples.

But what can you do about attack. You can say denied the attack or denied the man or denied the outcome. Those are all plausible but just haven't occurred in the training corpus. So why not take-- steal some of the probability mass and give it to the words that haven't occurred to generalise better. So if instead of saying, the probability of allegations is three, we'll chop it off and take off half or some amount-- take off half from allegations, reports, and claims. That gives you two to play around with. And that two, you give it to all the others.

So you've still got, overall there are seven. Something denied the somethings. But we're sharing out the probabilities amongst the other words. That's what smoothing basically means.

Now Laplace came up with this idea of add-one estimation. And it's a rather, a crude way of doing it. And it works reasonably well. And in reality, people use more complicated methods. But let's just try this. So if we pretend that we saw each word one more time than we actually did, so that's-- if you've seen the 15 times, then we just say we saw 16 times instead.

Or in fact, even in a large corpus, we might have said the 900 times. But we'll say we saw 901 times. The difference between 900 and 901 isn't that important. If we say-- if we never saw a word, and we say we saw it once, then it becomes significant. So we just add one to every frequency count. So the language-- so the estimate is instead of saying in a language model, the Markov model language model, we say the probability of any word, given the previous word, is the count of that pair of words divided by the count of the individual word.

So with add-one estimate, we say, you count up the pair of words, add one to that, and divided by the count of the frequency of the word. And since we're adding one to every word in the vocabulary, we have to add the vocabulary size. And that gives us estimates of everything. The maximum likely estimate of some parameter is something which maximises the likelihood of a training set, given language model M.

So suppose the word bagel occurs 400 times in the million word corpus, say the LOB or the Brown corpus. There's 400 occurrences of bagel. We just say-- what's the random word-- what's the probability that a random word from some other text will be bagel. Well, the standard Markov language model estimate is 400 divided by a million, which is 0.0004. That may not work for another corpus, because by poor chance, it just doesn't have the word bagel in it.

But it's an estimate. And it's the only estimate we've got. Because we've only got the training corpus. OK, so the estimate, if we're looking at, say, the Koran or the Bible, it probably doesn't have the word bagel in it. But if we've only got this training corpus, then the estimate has to be 400 divided by a million or 0.0004.

Now let's look at the Berkeley restaurant corpus again. Remember, this is one where we have I, I occurred occasionally. I want occurred occasionally. I, I occurred five times. So now we have this smoothed bigram corpus with Laplace says, just add one to everything. So there's no longer any zeros. I want now occurs 428 times. Or I, I occurs six times, rather than five times. We've also got want, want rather than occurring zero, occurs once.

Now if you add one to everything, then you no longer have any zeros. And that means you can now calculate probabilities of everything. And there are no zero probabilities. So I, I has got quite a small probability of 0.0015. Remember that was now-- the count of I, I plus one, which is now 6, divided by the count of I, which is quite high, plus the vocabulary size. So 6 divided by quite a large number gives you 0.0015.

So I, I is now not likely but not impossible. But also want, want, suddenly it becomes, instead of being zero, the count of want, want is zero. But it becomes one, divided by how many times did want occur, quite frequent, plus the vocabulary size. So want, want has a non-zero probability. Notice all these zeros are replaced by small fractions but not the same small fraction. It's still dependent on how frequent the word is.

So want is a quite frequent word. So want, want has got a reasonably high probability, whereas Chinese has a different probability and so on. And this then in principle, we can if you like, reverse engineer the frequency counts. If we wanted to-- having got these probabilities, then you can work out the likely count, given the Laplace probabilities. So now suddenly I has a count of 3.8, if we're reconstituting. Some of these counts are non integers. But even want, we see the likelihood-- the count of want, want is 0.39.

So some of these counts are actually lower. The count of I, I is less than what we actually counted. Because we've taken away the frequencies and redistributed them to the zeros. So you can-- here we can see the raw bigram counts and the reconstituted bigram counts side by side. Whereas I, I actually occurred five times, in the reconstituted count it's only 3.8 times. I want is drastically lower. I want was 827 times. In the reconstituted, it's much less. I want is only 527 times.

This doesn't really matter, because I want is still very high. So we're not saying I want is unlikely, we're just saying it's less likely. But the advantage of this is that we no longer-- we don't have any zeros. We have very small fractions, instead of zeros. But these very small fractions means you don't divide by zero anymore. Or rather when you have a chain probability, there's no zeros in the chain.

So if you're multiplying things together, if one of the items is zero, then the whole thing becomes zero. The whole sentence has a value of zero, whereas now it has a sentence with a small fraction in it. We have a low probability but a non-zero probability.

Add-one estimation is a blunt instrument going back to the idea of hammering things around. And it's not actually used in state of the art n-gram language models, because there are better methods. But they tend to be much more complicated. So if you're doing n-gram modeling, using one of these n-gram models that I've shown before, then it uses more complicated systems. This works-- add-1 estimation is particularly good for some language models where it doesn't matter too much.

For example, if you're building a classifier, all that matters is that the best sentence is the right one. So rank order is sufficient. Getting the exact numbers you generate don't matter too much, as long as the right one is first. And the bad one is second. OK, the rank ordering is sufficient. Also, if you're just dealing with the Koran or the Bible or a small corpus on a limited domain, then add-1 estimation may be sufficient.

OK, that's the end of add-1. Finally, I just want to quickly look at another way of handling zeros. And that is backoff. Sometimes you can use less context. So if your trigram model has a zero in it, you don't have any idea. Then why not use the bigram model instead. Backoff is use the trigram model, if you have sufficient evidence. If not, then use the bigram model. If the bigram is also zero, then use the unigram. That will do instead. That's a different way.

And you can use backoff, even if you're using Laplace estimation, because sometimes you still have not very good values. This backoff is used as an alternative. If you have a good, solid estimate of a trigram value, then use that. If you're not sure or the bigram is zero or very, very small then-- sorry, the trigram is zero or very small, because it hasn't occurred in your training set. Then use a bigram model instead. Otherwise use the unigram.

An even more sophisticated method is interpolation. So you can mix up interpolation works Better, but it's a bit more complicated. For interpolation, what you do is you take the trigram model value and also-- sorry, some constant lambda 1 of the trigram probability plus some constant tri-- or lambda 2 of the bigram probability plus some constant lambda 3 of the unigram probability. And the lambdas one, two, and three should add up to one.

So it might be, I don't know, a half of the bigram of a trigram probability plus a 1/4 of the bigram probability plus a 1/4 of the unigram probability. And that way if your training corpus is not very big, and that means the trigrams are overfitted but the bigrams are less fitted. And the unigrams are even less fitted. Then this is a way of avoiding overfitting by taking into account the less fitted unigrams and bigrams.

Even more complicatedly, you can condition, the lambdas can be conditioned on the context. And I'm not going to go into this now. But have a look at the textbook. If you're very keen on the mathematics, the textbook explains this in more detail. But all I'm trying to show you is that if you use interpolation, then it gets complicated. And the more complicated it is, the less fitted you are to a training corpus. But the more computation there is. And for web scale things, it gets complicated.

So how do you set these lambdas? Well, you want to try to avoid being too fitted to a training corpus. What you can do is chop the training data into a large subset and a small subset, which is still in the training data. And you call that held-out data. So you have a training corpus, a held-out corpus, and a test corpus, which you don't touch at all. And you choose lambdas, which maximise the probability of the held-out data.

First of all, you train the n-grams on the training data. And then you use the held-out data to optimise the lambda, try different lambdas on the held-out data. And that then uses-- that means you're using the training data and held-out data as training data in total. But you're using the subset training data for calculating the n-grams and the held-out data for optimizing the lambdas. And then whatever model you get overall, you then use that on the test data. It is complicated.

You have the problem of unknown words. This is the real problem. These zeros come about, because there are words which are not in your corpus. Sorry, words which are in your training corpus can be different from the words which are in your test corpus.

If we do know all the words in advanced, then the vocabulary is fixed. And you have a closed vocabulary task. For example, if you're building a model just for the Koran or just for Shakespeare, and you're never going to use it on anything else, then you can assume that you know what the vocabulary is. You've just got all the words in the Koran, all the words in Shakespeare.

But often you do want to train a model on something and then use it for something else. And then you have Out Of Vocabulary, or OOV. Out-of-vocabulary words, in whatever you're using it on, you

have an open vocabulary task. What can you do about that? Well there is a sort of-- out of-- a trick is to create an Unknown word Token, or UNK, and then if you have a fixed lexicon of size v, when you are normalizing it and decide that you're training on any training word-- if you have a very large training corpus, and you have a fixed lexicon, so you're only going to use words in the lexicon. Whenever you come across a word which is not in the lexicon, then label it as UNK.

Then we can train the probabilities, just as if it's a normal word. So basically, you decide I'm going to use, let's say the 1,000 most common words in English. My training corpus, I go through the training corpus, whenever it's not one of those words in the top 1,000. But now label it as UNK instead. Now, I've got lots of occurrences of UNK in my corpus. And I could just use that as a sort of word.

And then also in the test corpus, whenever I come across a word which is not in my one thousand words, I label that as UNK. And the UNK in the test corpus may be a different word from the UNK in the training corpus. But it doesn't matter, because now it's just called all UNK. So in text input we use UNK probabilities for any word that's not in the training corpus.

OK, that's a cludge. It's not very realistic, because what we're doing is we're ignoring everything, apart from the 1,000 most common words or the ‘n’ most common words. And you don't really want to do that, because you're not really capturing the meanings of lots of words. Another problem is for web scale n-grams. So how do you deal with, for example, the Google n-gram corpus?

Well, one thing you could do is only store n-grams which have a count above a threshold. So remove all the ones, singletons or things that only occur once or things that only occur less than 20 times. And they've already done that. If you download the n-gram corpus, they've already said, we're only interested in things that occur at least 40 times. But you could say, I'm actually only interested in things that occur at least 100 times or 1,000 times.

Another thing you can do if you're dealing with web scale n-grams, is try to use efficient data structures, like "tree," or "try," depending on how you pronounce it, or bloom filters. You may have come across some of this in your algorithms module. Another thing is, OK, remember I said, you can use backoff. If you found something in your trigram or your 5-grams, then use the 5-gram. If the four words before are not in the 5-gram model, then use the 4-gram. And if not, use the trigram and so on.

Don't try interpolation, because that's more complicated. Just use backoff. Another thing very often is you want to store each word as an index, rather than the actual word itself. And you can use clever Huffman coding for example, to store words as bytes, rather than whole numbers.

Google have done some interesting experiments to show that instead of using a whole 8-byte float number, you can just use eight bits or even four bits as a sort of cut down number. And that works reasonably well. So there are various tricks that Google has done to compress, not just the data and the n-grams, but also compress the storing of numbers when you're doing calculations. You could have an 8-bit code for a number, instead of using a whole float.

And Advanced Language modeling involves using adaptive models, choosing n-gram weights to improve a task, rather than just using it from a training set, or even parsing-based models. That is, take the grammatical structure, take different language models depending on whether it's a noun or an adjective or a verb. Or you can use caching models. Recently used words are more likely to appear.

So training corpus doesn't take into account anything about what you're seeing. So for example, if you're processing text on the fly, then adapt the language model according to the words you've just seen recently to increase the likelihood of words you've seen recently. This is another thing you can do.

OK, that's the end of web scale language models for now. And that's all I want to say about n-gram language modeling. We've looked at an introduction to why you should use n-gram models. We've looked at how to estimate the probability of n-grams from a training corpus. Ideally having got a language model, you want to train it on a training corpus but then test it on an actual task, like machine translation. If you can't do that, then you can measure perplexity in some way.

There is a problem about generalization that a language model training or training corpus may not be appropriate for a different test corpus. The language of Shakespeare is quite different from the language of the Wall Street Journal, for example. There's a real problem with zeros that very often a word or a bigram or a trigram in a test set may not have appeared in the training corpus at all, which gives-- since you're working with probabilities, that basically says the probability of a sentence is zero, if you find one particular part of it, which you haven't seen before.

So to get around that, we try smoothing. And add-one or Laplace smoothing is very simple. You basically, whatever you're counting, add one to it. And if you've counted 900, then make that 901. But if you counted zero, make that one. And that has-- that works to some extent.

Another thing you could do is backoff. If you have a trigram which you haven't seen before, then try the bigram. And if you haven't seen that try, the unigram. Or you can try interpolation, which means combining trigram, bigram, and unigram in a clever way. For web scale language models, backoff or various other clever fixes are available too, and more of that in the textbook.

OK, thank you for listening. I now recommend that you go and read the textbook for more details. Chapter 3 has got a lot more in there. But I've given yourself an overview of what to expect. Thank you very much and goodbye.

[END]