Text pre-processing

Professor Eric Atwell: OK, so this is a summary of Chapter 2 of Dan Jurafsky and James Martin's 2022 textbook on speech and language processing. We have provided the PDFs of the textbook for you to read. And if you want to buy the original textbook, the 2022 edition is isn't currently available as I speak, but hopefully it will be later in the year available.

OK, so the first issue in text mining as opposed to data mining is to do with the format of the data. Typically if you're dealing with machine learning from data on a machine learning course, you've probably been given instances where each instance is a sequence of numbers separated by commas in CSV format or something like that-- a spreadsheet, effectively of numbers. Whereas most text is a sequence of characters and when you read it, you as a human can see it's actually made up of words and sentences and paragraphs. But the computer can't do that.

So one of the things you have to do is process it, or pre-process it in some way, to tokenise it or break it into chunks, to recognise sequences of characters which are bigger units. And also to recognise two sequences which are essentially the same, but have minor variations. So you can join them together.

And this is what regular expressions are for. So regular expressions are a formal language for specifying patterns of text strings. So for example, if I want to capture woodchuck in its various different forms, then you can use this thing called disjunctions. A disjunction, if you've done the AI course on knowledge representation, then you'll know a disjunction means or.

So if you put little w and capital W inside square brackets, that means either little w or big W, followed by oodchuck, and that will match woodchuck stock and capital letter, woodchuck all in lowercase. Or you put open square brackets 1234567890, close square brackets. That should match any digit.

Now regular expressions are available in many different programming languages like Python, or Java. They're also available in Linux, or Unix, or OS X as part of the command language. The exact format, the exact characters that are used for representing these patterns may be slightly different. So I'm showing you the generic one that's used for Unix.

You can also have ranges. Like open square brackets, a dash z, closed square brackets, matches anything in the range a to z, any uppercase letter. You can also do that for lowercase letters, or for digits. A bit more complicatedly, you can have as well as disjunction, or, you can also have negation, not.

So the carat or up arrow symbol means not. So not capital S, little s means any letter or any character other than S. Or not A to Z means any character other than uppercase letter. So it could be a lowercase letter, or there could be some other character. It could even have carat, e, carat, which means any character other than e or the carat character, and so on.

So there are lots of patents involving that. You can also have what we call a pipe letter or character, which is another way of saying a disjunction, or or. So in America, if you're in America, you might know that woodchuck is another name for groundhog, so if you want to match groundhog or woodchuck, then you use Groundhog and then the pipe, and then woodchuck.

You can add yours, or mine, or a or b or c. Or it can be even more complicated. You can have little g big G in square brackets, roundhog, or little w, capital W oodchuck, and this will match woodchuck, groundhog, depending on whether or not it's a lowercase or uppercase as a first letter.

More regular expressions-- you can have these spatial pattern matching characters. Question mark means an optional previous character. So c-o-l-o-u question mark r-- that means u is optional. So it'll match the American spelling of color and the British spelling of color.

You can also have the star. So o, o, star h exclamation mark, will match zero more of the previous character that is o, followed by zero or more O's, followed by h. So I match all of those things. The plus means one or more of the previous characters. So o plus h is actually equivalent to o, o, star, h.

And these operators, star and plus, they're called Kleene operators, after Stephen Kleene who thought them up in the first place. And you can have plus b, a, a, plus, means baa, or baaa, or baaaa, and so on.

You can also have the dot as an operator, and b-e-g dot n, dot means matching any one character, such as begin, in or begun, or even beg 3n. Other more slightly more sophisticated characters at the regular expression patterns are the anchors. So the carat we saw before means not, if it's inside a pattern. If it's outside a pattern, it means the beginning of a string.

So carat and then A to Z means any string where the initial character is a capital letter. Well here we have carat, and then open brackets, carat again, A to Z, a to z. That means the start of a string, and it's not A to Z or a to z. It's not a letter. So for example, 1 will match that, or the quotes in quotes, "hello" quotes will match that.

Now because there are these special characters like dot, which have a pattern, meaning, if you actually want to use dot full stop, then you have to a backslash in front of it. So backslash is another special pattern character which means literally use the next character rather than using it

as a pattern. Whereas dot on its own means any character at all, dollar means the end of a sentence. So dot dollar means the last letter of a string.

And I'm to look at some examples. If you've done Python programming on string processing, you've probably seen this before, so I won't labor this. And if you want some more examples, look at the textbook. But for example let's say I want to find all the instances of the word the in a text.

One obvious thing to do is the character pattern, t-h-e. But the trouble is it will miss out t-h-e if it's a start of a sentence. So why not say open square brackets t, little t, capital T, and that will match t-h-e as we did before, but also the is at the beginning of a sentence. But unfortunately it will also match other or theology.

So here's a more complicated pattern. The thing in blue is not a-z, A-Z. That means not any pattern, which-- it must have a letter of the alphabet followed by lowercase t or uppercase T, followed by h-e, followed by not any other letter of the alphabet.

So other is ruled out, because you've got o, which is a letter of the alphabet before the, and R after the. And theology is also ruled out, because you've got-- OK, you haven't got a letter at beginning, but you do have a letter after the. So this is a pattern, which hopefully matches more examples of the and rules out other and theology.

And that works. This general process is something you have to do very often in machine learning or data analysis. You want to match strings that otherwise should not have matched. So we want to avoid the error of matching the things like other, or there, or then. These are called false positives.

So if we just look for t-h-e, then you're going to match other. And that's a positive, means to match, which is a false positive, because it's not actually what we wanted. So we have to look out for false positives.

The other thing we're trying to avoid is false negatives. That is ruling out cases that we do actually want. So if we, with a very simple pattern of just t-h-e in lower case didn't capture the at the beginning of a sentence, because it hasn't got a capital letter.

So that's saying capital T-h-e is a negative, and we've ruled it out. So that's a false negative, because we do actually want it. So false positives and false negatives are two types of error, and you get this a lot in any sort of classification task in data mining and machine learning.

In natural language processing, or text and analytics, we're often trying to rule out these types of errors. And if you rule out too much of one sort of error, you risk having, allowing, the other sort of error.

And these two sorts of errors are antagonistic. That means they pull in opposite directions. If you want to increase accuracy or precision, that means you want to minimise the false positives. You want to get a high accuracy-- anything you predict as being the, you do really want it to be the.

On the other hand, you may want to increase recall. That is all the real examples of the, you want to capture them. So that's minimizing false negatives. So recall and precision are different metrics and they pull in opposite directions potentially.

OK, so in summary, regular expressions are surprisingly important in text analytics. You often have quite sophisticated sequences of regular expressions, and they can be quite good at doing some sort of text processing. You don't really need machine learning, except for more sophisticated problems.

You can get away without doing any machine learning by doing quite complicated regular expressions, and that will match quite a lot of tasks. However, for harder tasks we do need to use machine learning classifiers. But even then, regular expressions can be used for pre-processing the data, or possibly for capturing the features that we want in the classifiers, or for capturing generalizations, even if you want to do machine learning. And this is before deciding whether you want to do deep-learning, or traditional, sort of simple machine learning.

OK. So that's enough on regular expressions. Except let's have a look at some examples. And we want to have a look at some examples of substitutions is one more regular expression that's often available in Python and Unix commands, that is having got a regular expression, you want to substitute it with something else.

So for example, c-o-l-o-u-r is a regular expression matching just the string c-o-l-o-u-r. And maybe we want to Americanise all the text like substituting all the British spellings of color with the American spelling or color. Or we might want to say, whenever we come across a number, put it in angle brackets.

So we want the regular expression open square brackets 0 to 9 close brackets plus, means any sequence of matching patterns where the pattern is a digit. And we can use backslash 1 to refer to whatever it is it's matched. So we have a register matching.

So for example, the pattern at the bottom is in brackets, any-- the pattern in brackets is any sequence of 0 to 9, any sequence of digits, and we replace it with open angle bracket, whatever the pattern was, followed by closed angle bracket. So that's a nice way of converting the 35 boxes into the open square, open angle brackets, 35, close angle brackets, boxes.

We may have more than one register if we have more than one pattern. So for example, the pattern the, dot, star, er, which means that match anything followed by er, they something, dot,

star. So any pattern which has the, something, er, they, something, comma, the, something, we, something.

So the faster they ran, the faster we ran. That will match. But it won't match the faster they ran the faster we ate, because ate has to be the same as ran, and it's not. Could also have even more complicated patterns called look-ahead assertions. And if you look at the book, you'll find some more explanations of this.

These are actually not very commonly used, but here's an example. If you want to match at the beginning of a line any single word apart from volcano, then here's a pattern which matches. So A to Z, a to z, is any sequence of characters which will match any word at all, but in brackets we said we don't want a volcano. So this is a sophisticated pattern for matching any sequence of characters other than the word volcano.

I'm not going to spend any more time on this, because it's pretty unusual to want to do that sort of thing. If you do want to, then look at the textbook for some more examples. OK, here's an example of an application. Later on in the module, we'll look at chat bots in general. And the very famous example of a chat bot is ELIZA.

Back in 1966, I was just a lad, there were it decided a very potential, simple, but useful use of chat bots is to talk to people about their mental health problems. And a Rogerian psychotherapist is someone who believes the teachings of Roger, who thought that basically what the psychologist has to do, or psychotherapist has to do, is talk to the patient and get the patient to draw out for themselves what their problems are.

So you encourage them to talk about their problems. So a pattern the user or the patient might say, I need x where x is something, and then the psychotherapist replies by getting to draw out more about what they want, what they need. It says, what would it mean to you if you got x?

So a lot of the things that the psychotherapist does is not introduce new information, but simply draw out on what the patient said. Repeat it back at them in some way a bit more convolutedly to get them to talk more about it. Here's some examples.

Men are all alike. Well you could come back saying, what do you mean about men? But if you haven't got-- if you don't want to be too specific, you can say, in what way? Or they're always bugging us about something or other. Can you think of a specific example?

Or you could even repeat back what the person said. Well my boyfriend made me come here. Your boyfriend made you come here? Well it's not repeating exactly back. You changed your mine to your, and me to you, but it's more or less what you said.

He says I'm depressed much of the time. And then you can repeat back depressed, but say, I'm sorry to hear you are depressed. So repeating back part of what was said using some sort of patterns.

And here's a simple pattern. I'm-- and the pattern is depressed, or sad, or some other word like that-- and you just repeat back I'm sorry to hear you are, whatever it is the pattern was, depressed or sad. Or slightly more complicated example is, star always star.

In other words, if they say anything followed by always, followed by anything, you come back with can you think of a specific example? So these simple patterns are all that ELIZA has. You can try this out yourself later on in the course and build your own ELIZA-like pattern systems. You can say even more complicated things in ELIZA, but we'll come back to that later.

OK, so we want to try to collect a corpus. A corpus is a text data set. And typically you want to split it up into words, and do things like counting up how many words there are. Well you might think counting the number of words in the sentence is easy, but it depends on what you mean by a word.

Here's an example. I do uh main- mainly business data processing. How many words is that? Well, part of the problem is, maybe there are fragments, like main- mainly. Particularly in transcriptions of speech, and maybe ooh, and ah, and pauses and interruptions and things like that. Do you want to count those? Well probably not.

Another thing is OK, here's another example, Seuss's cat in the hat is different from other cats. Well in this example, maybe cats is the same word as cat. Do we want to count those twice, or once?

So a lemma is the same root or stem with the same grammatical category. It's a noun in both cases, it's just once the singular once the plural. So maybe if we're counting how many words there are, how many different words there are, is different from how many individual words there are. So cat and cats are different word forms, but the same underlying meaning in some sense.

And this brings us to the difference between word type and word tokens. Here's another bit of text. They lay back on the San Francisco grass and looked at the stars and their, blah, blah, blah. So a type is an element. A token is an instance of that type in running text.

So we might say, the word the appears twice here. A more formal way of saying this is, there are the type the appears as two tokens in this example. So how many words are there? Well there's, they lay back on the San Francisco grass and looked at the stars and there.

That's 15 tokens. Or we might say San Francisco is really one word, in which case there's 14 tokens. But then how do you decide what a word is? Well because there's a space in there, well

maybe you have a dictionary of all the words and the dictionary has San Francisco down as one word.

However there are repeats of the and-- any other thing that's repeated-- and is repeated twice. Therefore there's not 15, but 13 is two less, because the and, and are repeated. Therefore there's 13 types. Except you might say there's San Francisco's one word, so that makes 12 types.

Or well there may be other things. Maybe you might say they and there are basically the same word, but different grammatical-- that they and there are-- they're both the same pronoun, but different forms. So maybe there's only 11. So it depends on how you divide up words and group words together.

In general, there's a sort of rule that if you take n as the number of tokens and v or the vocabulary of a dictionary might extract from that as a set of types, then the size of the vocabulary is different from the number of tokens. And there's a sort of law sometimes called Heaps Law, Herdan's Law. It's not actually a physical law in the same sense, it just says there is a general relationship that perhaps more vaguely, the vocabulary size grows as the number of word token grows. And it's usually greater than the square root of a number of tokens.

So in a larger-- if you get more and more text, you'll carry on getting obviously more and more tokens. But also the number of types will also grow, but not as fast as the number of tokens. And we can see this in a number of examples.

So for example there's a switchboard phone corpus, which is lots of phone conversations in America. And they've recorded them and transcribed them. And it's 2 and 1/2 million tokens altogether so that one way of measuring the size of a corpus isn't so much in how many bytes or megabytes or terabytes it is, but how many words there are.

That's the usual way of counting the size of a text data set, is how many word tokens there are. And there's 2 and 1/2 million tokens. That's the size. And there's about 20,000 different types.

Now if you look at the complete works of Shakespeare, that's less than a million word tokens-- slightly less than a million, and there's 31,000 types. So actually Shakespeare is more productive. It's more varied. That's completely not surprising.

When people talk on the phone, they tend to talk about the same sorts of things over and over again. So there's not as much variety in what they say as in the complete works of Shakespeare. Maybe more surprising, one of my first jobs a long time ago was to work on the LOB Corpus-- The Lancaster-Oslo-Bergen Corpus of British English, which is like the British English equivalent of the Brown Corpus of American English.

This was supposedly about a million words of text. At the time the internet didn't exist. So we had to collect published newspapers, books, other documents like that which were available. And had to be typed up again and transcribed-- well not-- typed up onto the computer.

So there's about a million words. And this contained 50,000, so that was more. So British English at the time in the 1960's and American English at the time in the 1960's was more varied than the works of Shakespeare.

And this is because, OK, Shakespeare was very creative, but it was just Shakespeare. Whereas the LOB Corpus and the Brown Corpus, that included newspapers written by lots of different people, novels, books of various sorts, government reports. All sorts of things written by lots of different people, and therefore it was more varied, because there are more different sources around.

Just by way of contrast, Google has access to the entire world wide web. So they have a huge amount of text data. That's why if you want to do research on text, it's a good idea to go and work for Google research labs. They actually collected the whole of the web and from it, extracted all the Ngrams.

An Ngram is a sequence of N words. So unigram, or one gram is just one word. A bigram is two words. A trigram is three words. A four gram is a sequence of four words, and they did it up to five grams.

So they collected all the sequences of one, two, three, four, or five, words which came together, which were found. And this whole corpus is based on a trillion that a trillion different Ngrams that appeared on the web, but there are only 13 million different ones. So there we see Heaps Law at maximum.

So it depends on where you get the words from. So a text is typically written by a specific writer at a specific time. And it might be writing a novel, or a newspaper, or some other variety. It may be English, or Arabic, or Chinese. And it may be for teaching purposes, or for instructional purposes, or to set the law, or for some sort of function.

So when you collect a corpus, you should record this as metadata. You don't just say, here's a million words, corpus. You have to say what is the language, what is the variety. What sort of-- maybe there are-- it can include code switching.

If you're collecting a Spanish corpus, then there may be English words sticking in it. When I supervise Arabic PhD students, and when they talk amongst themselves I can hear English words cropping up every now and again, because the technical computing terms tend to be in English.

It also depends, what is the genre or the type? Is it news, or fiction, or scientific articles, or Wikipedia? You also may, if you can, you want to take into account the demographics or where the author came from with their age, gender, ethnicity, and so on. This will affect what the vocabulary is likely to be in there.

So there are various guidelines on what to include. If you're collecting a corpus don't just collect the data, but you also have to have metadata about what's in it. And there's a couple of references to metadata guidelines. Why was the corpus collected? Who collected? Who funded it?

And certainly, if you want to find out these guidelines have a look at the-- read the chapter. You will find more information about this. So for your exercise this week, I want you to try and collect the corpus for yourself. So that's why I'm saying this. Don't just collect the corpus, but note down metadata about what features of the corpus are important.

OK, so that's word and corpus. Finally, I want to say something about tokenisation. So if you are collecting data and analyzing data for a classifier, you want to normalise it in some way. First of all, you need to segment the text into words, and then you also want to normalise the format of the words. You may even want to segment it into sentences.

So a very simple way to tokenise English and other language, which are written in characters like Greek, or Arabic, and so on. They typically have spaces between the words, so you can say wherever's a space, that segments it into tokens.

Unix has got a number of tools for a very simple space-based tokenisation. I'm going to show a simple example. Ken Church, one of the developers of Unix wrote a really interesting paper called, "UNIX for Poets." If you Google Unix for poets you'll find the paper. Or if you look in the textbook, you'll find more examples of it.

Given a text file you can write a single line of Unix command to tokenise it and extract the words from it, and extract a word frequency list from it. So given a text file. Let's see how can we output the word tokens in their frequencies. And you can do this in Unix, or you can do it in Mac OS. I've got OS X on my MacBook and it works for this, too.

So here's a simple Unix command-- tr minus sc, A to Z, a to z. And what this does-- angle brackets mean take the text, shake stop text, put it into this command. It changes all the non-alpha characters to new lines. I want-- and then once you've basically chopped the text up into words, and put each word on a new line.

And the next thing you want to do, to sort them. And this sorts all the words into alphabetical order. And then the next command is uniq minus c. And this counts up repeats of the same word. So it merges each repeated version of the same word, and counts up how many times it appears.

So you end up with, for example, 1,945 occurrences of capital A, followed by 72 occurrences of AARON and followed by 19 occurrences of ABBESS, followed by 5 counts of ABBOT. However, what it is doing at the moment is, it's still sorting and merging and counting individual character strings. It's not merging AARON in all capitals in with Aaron, where the first letter is a capital letter and everything else is in lowercase.

So you want to do a bit more sophisticated than that. And this for example is given Shakespeare's text. It will do the same thing. If we have the sonnets by William Shakespeare from various creatures we, blah, blah, blah, whatever he wrote, this will take the sonnets and breaking it up into words by replacing every sequence of alphabetic characters with a new line.

And then sorting it will put all instances of each of the same letter, and so all the A's will appear first, and so on. So that's a nice way of doing that. OK, for more examples of how this is done, have a look at the textbook.

Typically if you want to do counting or most processing, you want to merge upper and lower case so this is the original example you saw of a command. Take the Shakespeare text. Translate all the uppercase into lowercase. And then sort the counts.

So having got all the counts, you then have for every word how frequent it is, you may actually want to sort the count into frequency order so that you have the most frequent word first, and the second most frequent word second, and so on. So rather than just outputting for every word what frequency it is, the second sorting sorts all these word counts into frequency order.

So we have the is the most frequent, followed by I, followed by and, and so on. And with your frequency of all these. So we can see that's the word frequency. Notice that D is on its own.

That's because the original-- if you had something like I'd, I apostrophe D, what it's done is taken wherever there's a sequence of characters which are letters that count as a word. So I apostrophe D is counted as two words, the word I, and the word D. And this happens whenever there are what we call, enclitics. That is I'd, or you'd, or I'm. The muh or duh counts as a separate word.

So we can't just blindly remove all the punctuation. It's not a good idea just to take out all the apostrophes, or the full stops, or so on, because there are some words which include it. Like miles per hour, or mph, is a word. Or PhD could have dots in it.

There are also amounts of money, like the dollar sign should be included in 45 dollars, fifty-five cents. Or dates, or URLs, or hash tags, or email addresses, we want to keep in as one word. And clitic is another example I just mentioned. We're, we apostrophe re, really means we are. Or in French-- anybody here out there speaks French-- j'ai, means I have, it's actually je ai. Or l'honneur means the honor, and it's actually means le honneur.

And the other way around. So we're is actually two words, even though there's no spaces in there. And the other way around, New York is really one word even though there is a space in there. Or rock 'n' roll, it appears to be rock n roll, but actually it's one word, a multi-word expression. And we'll see more about multi word expressions later on in the module.

There are lots of tools for doing tokenisation. I won't go into this in more detail now, but later on we'll have a look at NLTK. That's the natural language toolkit. It's a whole collection of Python code for doing natural language processing. You can use this for in your projects, or in your coursework later on.

There's a simple example. It allows you to use the sort of patterns we looked at earlier on and includes them in the Python so that you get, for example, if you have the text is, that USA poster print costs $12.40. It correctly tokenises those into the words that USA poster print costs $12.40, and so on. So it gets the tokenisation right, like as I said before, combining a sequence of these patterns.

So these regular expressions are very powerful, particularly if you can combine them in this far more complicated way. So you have one pattern, or another pattern, or another pattern, or another pattern, and that way it any word-- if it matches-- if it's just a character sequence it matches, or if it's one of the more complicated patterns it matches, and so on.

And this doesn't work for all languages. So for example, Chinese, Japanese, Thai, they don't have characters in the same way as English, or Arabic, or Spanish. Well how do you deal with this? Well in Chinese it's not so obvious what are the words. It's this idea that words are composed of characters. Each character is quite complicated and you might say on average a word can be one or two or three characters, but deciding what counts as a word is actually quite complicated.

Even for linguists, it's not actually agreed on. You might say if you're translating this into English-- I can't speak Chinese, so I'm going to try and read this out loud-- but it means something like, Yao Ming reaches the finals. Therefore we might say Yao Ming reaches the finals is 5 words. Therefore this character sequence must be 5 words.

But actually you might say on the other hand that, this sequence of-- Yao Ming is one word, reaches is one word, the Chinese doesn't really have a word for the, it's just final. So that looks like it's three words. However you might say Yao Ming is two words, it's not just one word. And finals is divided into overall finals, because overall finals means the overall last thing that you're doing.

So it depends on conceptually how you analyze the words. You can't just take the English translation and assume that if the English translation is 5 words, or 3 words, then the Chinese has 5 words. Because maybe conceptually Chinese thinks differently than English.

You might just say, there's 7 characters, so that just say there are 7 sorts of words. And the words in Chinese don't map onto English words straightforwardly. But after all, this is Chinese. Therefore maybe there are 7 characters. So in Chinese is common just to treat each character as a token, and in that case segmentation is very simple. But then how do tokenise words or meaning units, as opposed to characters, is a bit harder.

In other languages like Thai and Japanese, it's more complicated. What you can do instead, is if you have a training data set-- if you have a lot of Japanese text where some people have segmented it into words, then you can use this for machine learning, and standardly you might use some sort of deep-learning neural sequence model to learn segmentation. We're not going to go there yet. We'll see that later on.

There is in the book even more about text tokenisation using bytecode, and coding using single character segmentation of English, because if you say in Chinese every character is a segment, why not do the same thing in English? And there is a sort of way of doing that, but it's quite complicated. And I don't want to go into this yet, so we'll leave that.

Finally, I want to look at word normalization issues. So if you want to deal with text, sometimes you get U-dot-S-dot-A, and sometimes get U-S-A. You want to include them both as USA. Or even more complicatedly, the words am, is, be, and are, have the same meaning. They're all forms of the word be, so you want to tokenise them in the same way.

Information retrieval is Google Search. You may want to reduce all letters to lowercase, because you want to find matches for the word regardless if it's in lowercase or uppercase. But there are exceptions. The case can be important.

General Motors is the name of a company, whereas general motors in lowercase could just be any cars at all. Or S-A-I-L, SAIL, is a piece of software, whereas the word sail, s-a-i-l in lowercase is a different word. So for sentiment analysis for machine translation for information extraction, for most of the applications of text analytics, the case is important.

U-S in upper case is different from the word us, in lower case. You also have to have a dictionary of words, and you want to represent the words by their lemma. So for example, am, are, and is, you want to convert into be, or car, cars, car's, and cars' into car.

In some languages like Spanish, it gets more complicated. Quiero, and quieres, and querer, are the same. They all are different forms of want. And they can have quite a lot of different variations. In some languages like Spanish, and Arabic, you have many different variations of the same word. In English, not so many, but in other languages many more.

So he is reading detective stories. You want to normalise that into, he, be, read, detective, story. That's the aim. So these individual things, or routes, or morphemes, are the bits you want to get out of. The stem is the core meaning-bearing unit. And you can also have affixes added onto these.

So stem is the root, or the main core. And the affixes are things like s or plural, that you want to add on. These are both morphemes. Stems are morphemes and affixes are morphemes. So a morphological parser wants to parse the word cats into cat and s. Or for Spanish amaren wants to morphologically analyze that into amar, and n, but n is the plural and the future.

So it gets a bit more complicated. It's not just n But what is the underlying meaning is plural and future. We might say cats is really cat and plural, not just s.

Stemming is the bother of chopping off the affixes and just leaving the stems. If we want to say that cat and cats are both underlyingly cat, then you just want to throw away the suh. You don't want to-- don't need to know it's plural, just chop it off and forget about it.

Here's an example. This is not the map we found in Billy Boness his chest, blah, blah, blah. If you stem it, that gives us, Thi wa, not the map we found in Billi bone, because if you take in whenever you come across s, chop it off and throw it away. But unfortunately that will mean this becomes thi, and was becomes wa.

So maybe we don't want to do that. Stemming seeming simplistically, is a bit too simple. We have a more complicated stemmer available in NLTK and in Weka, as we'll see next week, and other things like that. Many other tools use this. It basically has a whole lot of rules for patterns which match.

So if you come across ational, or change it to ate. If you come across ing at the end of a word, then throw it away. Like monitoring becomes-- motoring, sorry-- becomes motor. If you come across S-S-E-S, then you throw away the E-S. But only throw away E-S, if it's after S-S. So grasses becomes grass. But lees, l-e-e-s, don't chop that off, because that doesn't have SS in it.

For more complicated languages, this doesn't work and you have to do even more complicated things, for example Turkish. Turkish is what's called an agglutinating language. That means you can link together lots of words into one word.

So here's a very long word. I'm not going to even try to pronounce it. But it means something like behaving as you are among those whom we could not civilise. This is a rather contrived example, but it shows in some languages like Turkish, and Finnish, and Estonian, you can get very long words. And chopping them up is quite complicated.

That's word segmentation. We also have the problem of dividing up the text in the sentences. Typically you could just say if there's a full stop, or a question mark, or an exclamation mark, that's the end of a sentence, and then chop it there. But full stop is a bit complicated, because it is all usually of a sentence boundary, but it might also mark an abbreviation, or it might be a part of a number, 0.02%, or other things like that.

So what you might do is use some sort of rules to decide if a period is a full stop or it's a sentence boundary. And you can do that by using the rules, using patterns, or you can have a machine learning algorithm. If you have a lot of text, and wherever there's a full stop some person has marked it's either a sentence boundary or part of the word, then you can use machine learning, deep learning, or whatever, to learn a way of predicting that.

It can help if you have a dictionary of abbreviations, like USA, or inc, or Dr. But often you don't need the machine learning. You can come up-- if you have a dictionary of abbreviations and rules to look for numbers, or percentages, then you can do that using patterns like that.

OK, look in the textbook for more about word normalization. I'm going to stop now. And I think now's a good time to go back into here and stop recording. Thank you very much for listening.

[END]