MITOCW | 2. Compression: Huffman and LZW

PROFESSOR: OK let's get started. Let's get started, please. All right, last time we talked about information and entropy. The picture we had was of some kind of a source emitting symbols. Symbols-- let's say n of them. So it chooses from these symbols with probabilities P1 up to Pn.

And then we talked about the expected information here, or the entropy, so the expected information you get when you see the symbol that's emitted by the source. And that was the average value of the information. So it was-- let's see, you take 1 over log of 1 over P i for each of the possible symbols. And then you've got to weight it by the corresponding probability to get an expectation. And this was the entropy of the source. Or if you want to make explicit the source, you could say H of S for source-- capital S. All right?

And then we were actually thinking of this operating repeatedly. So in the model we had last time, the source at each time chooses from one of these symbols with this probability. And it does it independently of choices at other times. So what the source actually generates is what's referred to as an iid sequence of symbols, independent, identically distributed. You'll see this a lot-- Or iid sequence of symbols.

So the independent part of this refers to the fact that it makes the choice independently at each time instant. The identically distributed means that at each time instant, it goes back to these same probabilities. It's the same distribution that it uses each time. So that's what iid means-- so sort of a stationary probabilistic source with no dependence from one time instant to the next.

Average information was measured in bits per symbol. And what we wanted to do was take those symbols and compress them to binary digits. OK, so we were going to-- you can compress them to other things. We were going to think of compressing them to binary digits because we're thinking of a channel that can take 0s 1s or a signal that's in two possible states. So what we wanted to do was take each symbol or sequence of symbols and code it in the form of binary digits. Right?

Now, each binary digit can, at most, carry one bit of information. If the binary digit is equally likely to be a 0 or a 1, then it carries one bit of information. So that tells you really that if you're going to code this, the code length-- let's see-- compress to binary digits, let's say, or encode. And what we need is the expected code length.

L should be greater than or equal to H. So you need to transmit at least this many binary digits on average to convey the information that's coming out of the source-- per symbol or per timestamp. All right, so that was the basic setup.

I've given you one of these bounds here. When we talked about codes, by the way, we decided that if we're talking about binary codes, we want to limit ourselves to what are called instantaneously decodable or prefix-free codes. And these are codes that correspond to the leaves of a code tree. So we had examples of this type.

You want your symbols to be associated with the leaves of-- the end of the tree, not intermediate points. The reason being that, as you work your way down to the tree-- by the way, I'm assuming that this picture makes sense to you in some fashion from recitation. But as you work your way down to the symbol, you don't encounter any other symbols on the way. So as soon as you hit the leaf, you know what symbol you've got.

So we're limiting ourselves to codes of that type because some of the statements I make are not true if you don't have codes of this type. So I won't comment on that again. All right, so we've got that, the first inequality that I've put up there. And it turns out that Shannon showed how to actually construct codes that will give you a band on the other side.

Let me actually write it the way it is on the slide. So Shannon showed how to get codes that satisfy this-- so can get code satisfying this. So Shannon showed how to get within one of the lower bound in terms of the expected length of the code. So that was pretty good.

But after coming up with this paper in '48 and working on this for a while, neither he nor other luminaries in the field had found how to get the best such code, and that's what Huffman ended up doing. So we've talked about that already. OK, so Huffman showed how to get a code of minimum expected length per symbol with a very simple construction.

Now, you can actually extend Huffman-- and maybe you talked about this in recitation as well. So you can code per symbol, or you can decide you're going to create super-symbols. Take the same source, but say that the symbols that it emits are the symbols from here grouped two at a time. So you're going to take the symbol emitted at some particular time and then the symbol at the following time and call that a super-symbol. And then take the next pair, and that's a super-symbol and so on. So you're doing the Huffman coding, but on pairs of symbols.

So you can go through the same kind of construction. If you assuming an iid source, then the probability of a paired super-symbol is easy to compute. It's just a probability of the individual ones because they're independently emitted. And then the entropy of the resulting source here turns out to be twice the entropy of the source here because these are independent emissions, so the entropies will just add.

So you can do the Huffman construction again. And what you discover is the same kind of thing except this is now the inequality, right? And the reason is-- well, here L is still the expected length per symbol. But you're doing pairs now, so the expected length for the pair is 2L. Right? The lower bound is the entropy of the source. That's 2H. The upper bound is the entropy of that source plus 1.

So you can construct a code of that type. You can do it with Shannon's construction or Huffman's. And now see what you've managed to do. You've got a little titer of a squeeze on the expected length.

So we've gone from H plus 1 to H plus 1/2 with this construction. If you took triples, this would just change to 1 over 3. If you took K-tuples, you'd get 1 over K. So if you encode larger and larger blocks, you can squeeze the expected length down to essentially what the entropy band tells you.

Now, Huffman-- you've spent time in recitation. I just thought I would quickly run through an example so that you have this fresh in your minds. So we start off with a set of symbols. This is kind of weak, but I hope you can it. A set of symbols, A through D in this case, with probabilities associated with it. The Huffman process is to first sort these symbols in descending order of probability. So that's what I really start with.

You take the two smallest ones and lump them together to get a paired symbol, rearrange, reorder. And then you do the same thing again. You take the two, combine them, reorder. Take the two smallest ones, combine them, reorder. And that's what you have for your reduction phase.

And then you start to trace back. So when you trace back, you can pick the upper one to be zero, the lower one to be 1. And then every time you get a bifurcation, as you go back, you'll pick the upper one to be zero and the lower one to be 1. And you start to build up your code word, right?

So this one traces back. There's no bifurcation. This traces back. The 0 becomes 0001. And you go all the way like that. OK? So trace back-- let's see. Oh, was there a-- yeah. So the 1 here becomes a 1 0 and a 1 1. And then at the next step, you're all the way back with the Huffman code. Right?

So that's the Huffman code for that set of symbols. It's a Huffman code. I shouldn't say the Huffman code because, if you notice, at various stages we had probabilities that were identical, like over here and over here and over here. And we could have chosen how to order these things and then how to do the subsequent grouping. And all of those will give you Huffman codes with the same minimum expected length.

All right. All right, I want to give you another way of thinking about entropy and why it enters into coding. And here's the basic idea. All right, so we're still thinking about the source emitting independent symbols. It's an iid source. And we've got a very long string of emissions. So we've got a very long string of symbols emitted, maybe S1 of the first time, S17 here, S2 here, and so on.

And the question is, in a very long string of symbols, how many times do you expect to see symbol S1? How many times you expect to see a symbol S2 and so on? Well, if you actually work it out, it turns out that the expected number of times, number of times we see SI in the [INAUDIBLE] symbol is K times the probability of seeing SI. So it's what you'd expect. All right?

So the expected number of times is that. Well, but that doesn't tell you what the number of times is that you'll see in any given experiment. We know that you need to think about standard deviations as well. So what this is saying is, for instance, for symbol SI, that we expect to get that many of symbol SI. But actually, there's a distribution around it. So you'll get a little histogram here.

I'm not making any attempt to draw it very carefully, but there's a distribution. You run different experiments, you're going to get different numbers of SI in that run of K. Right? So there's a distribution. And it turns out you can actually write an explicit formula for the standard deviation.

This is something you'll see if you do a probability course. It's actually very simple. So that's the standard deviation. So the standard deviation goes as root K. So the interesting thing is that the standard deviation is a fraction of-- the number of successes get smaller and smaller as K becomes larger and larger. Or another way to see that is, if I normalize this, so I'm going to do a number of successes divided by K. This histogram is going to cluster around P i. And the standard deviation now, because I've divided by K, the standard deviation actually now ends up being P1 minus P square root of K. All right?

So what this says is if you get a run of K emissions of the symbol and you try and estimate the probability, P i by taking the ratio of times SI appears over the total number of runs, you'll actually get a little spread here centered on P i. But the spread actually goes down as 1 over root K. So this is really what the law of large numbers is telling us. It's telling us that if you take a very long run, you almost certainly get a number of successes, well, Kp i in this case. It's very tightly concentrated.

All right, we don't want you to remember all these formulas. I have them on the slides. It's just there for fun. There's something else that I put on there that you can try out for fun. I don't want to talk through it, but you can use exactly this to analyze things like polling. Why is it that you can poll 2,500 people and say that I've got a margin of error of 1% as to how the election is going to turn out? Well, the answer is, actually, in exactly this.

If we have time at the end, I'll come back to it. But it's easy enough that you can look at it yourself. So let's focus on what it is I wanted to show you.

I picked Obama 0.55, but that was just as illustration.

[LAUGHTER]

No. No political views to be imputed to that. All right, so what we're saying is you've got K emissions of this symbol. And with very high probability, you've got Kp1 one of S1, Kp2 of S2, and so on. So this is really what you're expecting to get, provided you've tossed this a large number of times.

What's the probability of getting a sequence that has Kp1 of S1, Kp2 of S2, and so on? So you've got to get S1 and Kp1 positions. What's the probability of that? And you've got to get S2 and Kp2 positions. So how do you work out those probabilities? We're invoking independence of all the emissions. So you can multiply probabilities.

So what you have is S1 occurring with probability. P1 to the power Kp1, because P1 is the probability with which S1 occurs, and it's happening Kp1 times. So you take it to that power, and then P2 to the Kp2, all the way up to Pn to the Kpn. OK?

So this is the probability of getting a sequence like this. And what we've said is this is the only kind of sequence you're typically going to get. All the rest have very low probability of occurrence. So it must be that when I add up all these sequences, I get, essentially, probability 1. So the question then is how many such sequences are there. If a single sequence of this type has this probability, and the only sequences I'm going to get are sequences of this type effectively, and the probabilities have to sum to 1. How many sequences do I have of this type?

Do you agree that it's 1 over the probability? The number of such sequences? Because I've got to take the number of sequences times this individual probability has to come up to be one. Right? The number of sequences-- let me write this down. So that you see it a little better.

The number of such-- let me call them typical sequences-- times the probability of any such sequence has got to be approximately 1. I say approximately because there are a few other sequences whose probabilities might-- I would have to take a count of. But this is essentially it.

So the number of such sequences is 1 over this number. So the number of such sequences is P1 to the minus K1, P1, P2 to the minus Kp2 and so on. That's the number of such sequences. And essentially, these are all the sequences that I'm going to get.

Well, if I take the log of this-- just visualize how the log works. Now I've got the log of a product, so that going to be a sum of the individual logs. I've got the log of a power of something, so the power will come down to multiply the log. This comes out to be K times H of S exactly. OK, so the log of the number of sequences is K times H of S, K times the entropy. This is log to the base 2.

So the number of sequences is equal to 2 to the KH. I'm saying equal to. I should be putting approximately equal to signs everywhere, but you get the idea. So the number of typical sequences is 2 to the KH. How many binary digits does it take to count 2 the KH things? KH, right?

So what I need is-- so I just need KHS binary digits to count the typical sequences. So how many binary digits do I need per symbol? It's just that divided by K because I've got a string of K symbols. So I need a number of binary digits equal to the entropy. So this is a quick way of seeing that entropy is very relevant to minimal coding of sequences of outputs from a source like this.

All right, now I swept a lot of math under the rug. The math that makes this rigorous exists. We don't want to have any part of it here. But for those of you who are inclined, you can look in a book on information theory. There's a nice name to it. It's called the Asymptotic Equipartition-- Equipartition Property. OK? It's basically saying that, asymptotically the probability partitions into equal probabilities for all these typical sequences.

All right. So all that is for Huffman and its application to symbols emitted independently by a source over time. But there are limitations to this. We've been working with Huffman coding under the assumption that the probabilities are given to us. But it's typically the case that the probabilities are not known for some arbitrary source that you're trying to code for.

The probabilities might change with time as the source characteristics change. So you would need to detect that and recode, if you're going to do Huffman. And then the more important point perhaps is that sources are generally not iid. The sources of interest are not really generating independent identically distributed symbols.

What's perhaps more true is that-- let's see. Oh, here-- once you're done compressing your source to binary digits where each binary digit carries a bit of information, then you've got something that essentially is not correlated over time. You've managed to kind of decouple it. But at this stage, these symbols are not really independent in typical cases of interest.

So one important case, of course, is just English text. You can still code it symbol by symbol, but it's a very inefficient coding. If you wanted to do it symbol by symbol, let's just ignore uppercase. You've got 26 letters plus a space. So that's 27 symbols. Well, you could certainly code that with five binary digits because that would give you 32 things to count. You can do better with a code that approaches the entropy associated with a source of this type. That would be 4.755 bits.

OK, so if you ignored dependence in English text and just treated each symbol is equally likely, you'd say that that's the entropy, and you could attempt to code it with something approaching that. But actually, not all symbols are equally likely. If you look at a typical distribution of frequencies-- and we saw this with Morse already. The E is much more common than T, than A, O, I, N and so on.

So there is a distribution to this. But you can take account of that distribution and compute the associated entropy, and you get something a little bit smaller, 4.177 instead of the 4.7-something that we hadn't before. Because not all letters are equally likely. But this is still thinking of it symbol by symbol, not recognizing dependence over time.

But English and other languages are full of context. Right? If you know the preceding part of the text, you have a very good way to guess the next letter. Nothing can be said to be certain except death and-- well, you can-- in this case, you can give me the next three letters. Right? Anyone?

AUDIENCE: It's taxes

PROFESSOR: Taxes, yeah. So even though X taken in isolation has a very low probability of occurrence, if you look at the histogram on the previous page, you see that the probability is 0.0017. Letters are not independently generated.

Now, it turns out Shannon was actually one of the earliest to study this in experiments on his wife. He had her-- he presented her with bits of text from one particular book and asked her to guess the next letter and so on. And he had a 1951 paper that actually launched a lot of this, because he had developed now the tools for talking about it. His estimate was much lower than the 4-point-something. It was more in the vicinity of one bit, 1 to 1.5 bits. So there's a lot of compression possible with English text because there's this kind of a dependence here.

And just to tell you what it is that we're trying to compute when we compute entropy for these long sequences of symbols, we're sort of saying what's the joint entropy of a sequence of K symbols divided by K in the limit of K going to infinity. So this is what you might call H under bar. It's not over bar because I couldn't see how to do an over bar on my PowerPoint. But it's usually an over bar in the books. But this is really the object that you would like to get your hands on. For sequential text that has context in it, this is the kind of entropy that you really would like to be working with.

OK. So that brings us to an approach to coding that's really focused-- coding or compression that's really focused on sequential text. And this is the Lempel-Ziv-Welch algorithm that's in the notes. Turns out that Lempel and Ziv or Ziv and Lempel had two earlier papers. And then Welch improved on it in an '84 paper. And what's in blue over there is a bit of a mouthful. And each word kind of means something, so I thought I'd list it all there.

Maybe I've used too many of these words-- universal lossless compression of sequential or streaming data by adaptive variable length coding. And I'll come to talk about those terms on the next slide. And it turns out that this is a very widely used compression algorithm for all sorts of files. Sometimes it's for a part of it. Sometimes it's optional. Sometimes it's combined with Huffman, but all of these things that do compression pay homage to Lempel and Ziv at least.

They were also patented. Actually, Unisys owned the patent on LZW for many years. These have all expired now. But while the patents were held, it made for a lot of heartburn because there were things being done without knowledge of the existence of the patents. And then people got hit with lawsuits and so on.

Jacob Ziv, again part of this incredible heritage from MIT of people working here in the early days of information theory. He was a graduate student here at the same time as Huffman and many other people whose names surface in all of this. I was actually at an award ceremony of the IEEE, where Lempel got an award for his compression work. And people were given a whole minute for a thank you speech, a mini thank you speech. And everyone took their minute to mention this person and that and talk about the origins of the work. It's a lot to say in a minute but they managed to convey a lot.

Lempel came up and said, "thank you."

[LAUGHTER]

It seemed kind of fitting for someone whose life is devoted to compression.

[LAUGHTER]

I just couldn't help but crack up there. That was-- all right. Now the interesting thing about this is that there are theoretical guarantees that, under appropriate assumptions on the source, asymptotically, this process will attain that bound. Now the thing is, the word asymptotically hides many sins. Lots of things happen at infinity that are not supposed to happen. Or lots of things happen at infinity that never happen when you're watching. So the theoretical performance perhaps is not as important as the fact that it works exceedingly well in practice.

So we're going to talk a little bit about it. You've got a lab on it as well. So let me just say a little bit about what these words mean. So this is universal in the sense that it doesn't necessarily-- it doesn't need any knowledge of the particular statistics of the source that it's compressing. It's willing to take its hand at anything. OK?

So it doesn't need to know the source statistics. It actually learns the source statistics in the course of implementing the algorithm. And it does that by actually building up a dictionary for strings of symbols that it discovers in the source text. So it's built around construction of a dictionary.

What it then does is it compresses the text, not to things that we've seen here in Huffman, but to actually dictionary entries. So it's sort of like Morse's original idea, which was communicate the address in the dictionary rather than communicating the word itself or some compressed version of the word. So it compresses the text to sequences of dictionary addresses, and those are the code words that it sends to the receiver.

It's also a variable length compression scheme. But it's interesting that it doesn't take a fixed length of symbols to varying lengths of code words. It actually takes varying lengths of symbols to fixed length of code words. So it's a little bit backwards. But it's still a variable length in that sense.

So the way this works is that the sender and the receiver start off with a core dictionary that they both agreed on. And for our illustrations, we might say that they've agreed on the letters A through Z, lowercase A through Z. So what they have is these letters or this core dictionary stored in some register. Well, actually let me show you what it might look like.

So there's the register with, let's say, you have an 8-bit table. This is the dictionary that you have at both ends. So you can store 256 different things. And you've both agreed on what's going to go into those slots. So somewhere-- I think it's slot 97 in one of these particular codes, you've got the letter A. And somewhere else you've got B, and so on. Or the next position you've got B. You can store a bunch of standard symbols.

So we'll consider that all the single letter symbols are already stored in designated positions in this dictionary that's known to the sender and the receiver. So if the sender just sends 252, the receiver knows what 252 refers to because they've got that core dictionary that they've agreed on. Some of the text here, by the way, is stuff I've said already. So I'll actually go back.

And then what happens is that the source starts to sequentially scan the text that's arriving or that it's looking at and puts new strings that it's found into new locations in this table and then communicates the address for the receiver. The magic of this-- and I mean it's fiendishly clever, very simple, but very clever, is that the receiver can build up its dictionary in tandem with the transmitter building up the dictionary. It's just a one-step delay. So one step later, the receiver has figured out what that dictionary entry is.

So the transmitter or the source is building up the dictionary, looking at strings in the input sequence, communicating the address-- the addresses of the appropriate strings to the receiver, and the receiver is building up a dictionary in parallel. Now I think the easiest way to do this-- there's discussion in the text. There's also code fragments. But I think the easiest way for me to try and do this is to actually just show you how it works on a particular sequence.

And you may not get all the details all at once. I do have a little animation that I need to tweak a bit, and I'll-- well, it's not an animation, but a set of slides that'll help you understand, actually, this particular example. So I'll have that posted as well. But for now, let's just work through this and see what it looks like.

And I hope I don't trip over myself in the process. I hope you'll be forgiving. And I need these two blackboards to do it. OK. And I need some colored chalk. So what I'm going to have over here is the source. And over here is the receiver.

And the source wants to send a message that I'll put here-- A-B-C. This is going to look incredibly boring. But the algorithm does different things at different stages, so that keeps it interesting. And let's see 1, 2, 3, 4, 5. And then we hit a special case somewhere near the end here that is worth sorting out. Because otherwise that, the fragment of the code that you see doesn't make sense.

Gee, can you believe that I want to start this again here? Sorry. Let's start here. I want at least six replications of ABC. I want you to get comfortable also so you can settle into this. OK, here we go. All right. The receiver has no idea that this is the sequence. The source has, and the receiver both have A through Z sitting in their dictionary at designated locations.

So the source will first see the letter A and does nothing because A is in its dictionary. It doesn't want to do anything yet. Then it looks at-- it pulls in B. So now it's looking at AB. AB is not in its dictionary because it's a symbol of-- it's a string of two symbols. So now it knows it needs to make a dictionary entry.

I'm going to indicate dictionary entry with this. So the source is going to make a dictionary entry of AB. So what this means is somewhere in that register in a particular position, or in the next position actually from the agreed on table, it sticks in this.

And then what it transmits to the receiver is not this, but the code for A. OK? So it enters the longer fragment here as a new dictionary word and sends the address for the piece that the receiver sees. So what does the receiver get?

The receiver sees A coming in and says, OK, that's the sequence A. That's the symbol. A, I'm all set. All right? Now what happens is that the source pulls in the next letter. It's done with the A, so you can essentially forget about that. It pulls in the next letter. Looks to see if it's got B-C in its dictionary. It doesn't have BC because it only has single letter entries, and it has AB. So it's got to put in BC. So it's going to put in an entry for BC.

And then what it's going to transmit is the B. The receiver gets the B. Oh, sorry-- the directory entry for B. And so it knows that's the letter B. And now it enters AB in its-- in its dictionary, OK, in the next location. So you see, with a one-step delay, the AB that was in the dictionary here has ended up in the dictionary of the receiver.

OK, we're done with this. We now pull in the next letter here. That's A. We haven't seen A-- we haven't seen CA in our dictionary. So we make an entry for CA, ship out C. C comes here. I should say that this was done with the A. The C comes here, and the receiver knows to make an entry for BC. So with one delay it's got it.

OK, we're done with this. We pull in the next letter, AB. That's in our dictionary. So we keep going, all right? So this algorithm doesn't look to ship out the dictionary address every time it sees a sequence that it recognizes. If it's got this already in its dictionary, it keeps going to try and learn a new word. So it's already got AB there, so it keeps going and it pulls in C. And now that's a new word.

So it's got ABC as a new entry. It ships out AB-- the address for AB rather. This gets the address for AB, which is in its dictionary. It puts the AB down there. It takes the first letter of the string that came in and appends it to the last one that it had there and gives you the CA. So you see, it's keeping up but with a one-step delay.

Let's keep going. So the AB is done with. We pull in A. We've got CA. We pull in the B. We don't have CAB, so let's enter that as well. By the time we've done this example, by the way, I'm hoping you'll know Lempel-Ziv. So bear with me.

All right, dictionary entry-- and now what does it send out to the receiver?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Sorry.

AUDIENCE: C2

PROFESSOR: CA-- the address for CA, right? The address for CA. So the address for CA comes in. It decodes the CA. And so let's see. We're done with these pieces, but this one has to build up its new direct dictionary entry. And so what it's got is the AB setting from before, and it pulls in the first letter. Instead of wrapping to the next board, let me start winding up again-- winding upwards. OK, so that's the new entry there, the receiver-- one step delayed from here.

OK, I pull in the C. I have BC . I keep going. I pull on the A. I don't see that. So I need BCA. I ship out the address for BC. So I'm done with these. I get the address for BC here. I decode and get BC.

I combined the first letter of the new fragment with what was sitting here. So I get CAB as my dictionary entry. And I keep going. All right, it's very systematic. I'm going to keep going because there's a special case that will trip you up if you don't get to it. And we need to proceed a couple more here.

OK, I pull in the B. I've got a AB. I pull in the C. I've got ABC. I pull in the A. I don't have ABCA. So I enter that in my dictionary. And then I ship out ABC.

OK, so you're always building a new word, entering it in your dictionary, and then the part that's already known you're shipping out and then hanging onto the end of this to start building the new fragment. ABC arrives here. I had the BC from before. I pull in the first letter of that, and I get a BCA as my new entry, which is this one.

OK. Now we pull in the AB. I mean, we pull in the B. We have AB. We pull in the C. We have ABC. We pull in the A, we have ABCA, so we pull on the B. We ship out ABCA-- A-B-C-A. Right? And now we're done with all those guys. And here comes ABCA.

And I go to my dictionary, and I don't have ABCA-- big hiccup. So the reason that happened is that I'm discovering I need to send ABCA on the very next step after entering it in my dictionary on the receiver-- on the transmitter side. And so the receiver hasn't yet had a chance to catch up.

Now if you analyze this, It turns out that whenever this happens, the sequence involved has its last character equal to its first character. So looking at this, the dictionary here is waiting to build up. It's got the ABC here, and it's waiting to pull in the first letter from the sequence-- the sequence associated with this dictionary entry.

It doesn't have that dictionary entry. So it can't pull in the A like it was doing all along. But if you analyze the cases under which this happens, It turns out that whenever you don't have it in your dictionary entry, the missing letter that you want to pull into your dictionary is the same as the first one in that string that's waiting to be built up. So it completes it with an A, and it's all set. Now it says ABCA, and it continues

So this happens under very particular conditions. It's a special case. If you actually look at the code that's in the notes you'll see. While the encoding is straightforward, it's really remarkable that a short fragment like this can do that encoding.

Let's see here. I don't want to do this. I did another example. Let me just say what's on this before I dispense with it. Sorry. OK.

So look at what's happened. In terms of the number of things we've sent, we've only sent these addresses. And there are fewer of them than there were symbols in the original. So that's where the compression comes in. And as you get the longer strings, the benefit is higher. Actually, I'm going to pass this and just tell you, when you look through the code fragment for decoding, this is the special case that we talked about. If the code is not in your dictionary, then do such and such. So that's the explanation.

All right. And that's described in the slides. We'll put that on. I just wanted to end with a couple of things. One is actually-- LZW is a good example of something that you see in other contexts as well, where you're faced with transmitting data and you decide instead that you'll transmit the model or your best model for what generates that data. That can often be a much more efficient way to do things.

And in fact, when you speak into your cell phone, you're not transmitting a raw speech waveform. There's actually a very sophisticated code there that's modeling your speech as the output of an autoregressive filter. And then it sends the filter tap weights to the receiver. So this kind of thing arises again and again. Sending the model and the little information you need to run the model at the receiving end can be much more efficient than sending the data.

The other thing is everything we've talked about has been lossless compression-- Huffman and LZW. You can completely recover what was compressed. But there's a whole world of lossy compression, which is very important. And we'll find ways to sneak in discussion of that as well. All right, thank you.