UNIVERSITY OF LEEDS

Information retrieval and web search

Professor Eric Atwell: Hello, this is Eric Atwell, and in this lecture, I'm going to give you an introduction to information retrieval. Nowadays, it's commonly thought of as being Google search or web search. You type in some keywords, and there comes the pages you want. But there's a longer history of information retrieval where you type in some keywords, and you get the documents that you want.

So, in this lecture I'm first of all going to compare information retrieval with database querying using something like SQL, or structured query language. Then we're going to look at a particular format, the inverted file format, which makes the matching much more efficient between keywords and documents. I'm looking at two different models for doing this matching. One is a Boolean set theoretic model and then secondly, more realistically, a weighted vector model.

That's more like the model that Google and other search engines use. We're going to look at a couple of very simple worked examples to illustrate how this works and look at, how do you evaluate a search tool or an information retrieval tool? And then finally if you're not getting back sufficient, good answers, how do you do query broadening to improve the matching and get more of what you want and less of what you don't want?

So, let's first of all compare database querying with document search or web search. Well, in a database, the records are made up into fields. And a simple approach to searching for a particular keyword or string of characters would be to look for in any record, in any field, the string with wild cards before and after. So, if I want to find the word "graphics”, then I can search for percent graphics percent, where percent mean match anything.

But this is rather inefficient because it basically means we brute force scan to go through all the text data records until you find one or more of these. And this is because the information is stored in a structured manner. That's the whole point of a database.

So, you might say, well, if you want to find information, why not do information extraction? That is, rather than hold the full text, do the information extraction or content analysis, extract the named entities and keywords effectively, and hold these in a relational database. So, for example, the University of Leeds module catalogue, which has information about what modules are taught, what semester they're in, and other stuff like that, you can imagine having some sort of structured database which has modules indexed by index terms. And the modules would have information like the module code, the title, semester. The terms would have an ID and a value, and you'd have an index which links the module code to the term ID and so on.

And now imagine that you want to find all the modules which are related to a database or AI or knowledge base. So, to find all the modules, here's a bit of SQL to do it. And it's quite complicated already. You've a select distinct m from module...well, I'm not going to read it all out. But the point is you've got to find a term value is database or a term value is AI, or a term value is knowledge base.

And what about if you want to have a course which looks at...which includes database and AI and knowledge base? We might think just change the OR’s to AND’s to get that. That doesn't actually work because it's saying the T-value has to be database and at the same time a T-value has to be AI. That's not possible.

T-value can only have one value. It can't be two. It can be database or AI, but it can't be database and AI. So, this SQL doesn't work.

If you just want to say two terms, database and AI, then even that is quite complicated. What you actually have to do is to have two T's, T1 and T2 and specify when T1 value is database and T2 value is AI and find all the records or fields containing T1 and T2. So, it gets quite complicated if you want to search a database for several keywords...in other words, you want to find records which have got several keywords in them.

And that's why database querying using SQL isn't a sensible way. It gets much more complicated for "and" than it is for "or". And the trouble is in information retrieval, you typically do have several keywords, and you want to find all the documents which have all of them, not just a document which is any one of them.

So, instead of having a database stored in records and fields, you have a non-database structure. And therefore, it's not suitable for standard SQL. You have to think of every word, every index term, as a possible entry point to a document.

There are document record identifiers, and each document has a number of terms in it, a number of words in it. And what we actually want is for each word, a number of documents it contains. So, this is the standard indexing method for information retrieval systems. And it's more or less something that...something like this is used in most search engines. And that's called an inverted file.

So, the basic idea of an inverted file is, well, you start off with a number of documents. And document 1 will have term T1.1, T1.2, and so on. Document 2 would have term T2.1, T2.2, and so on. And what you actually want is for each term, a list of documents it contains...so for term 1, which documents that's in, for term 2, which documents that's in.

So, that's the notion of inverting. So, having got the set of documents, you then have to do some pre-processing to extract from all of a set of documents a dictionary of all the words that are in all the documents and for each word in the dictionary which documents it came from. And once you've got that dictionary, then you can much more efficiently find things.

Here's a dictionary, and the dictionary will have term 1, term 2, and so on. And it may have for each term how many times occurs. And then the next item 1 is a link into where in the postings file, so index where it is.

So, let's say for example we want to find examples of term 1, which might be the word "a”. And there's two examples of the word "a”. We start it post position number one. And since there's two examples, it must go from one and two and then stop at three.

And then we could just follow, where does 1.2...well, data file number 1 and 2 is data file number 2, so document 1 and document 2 have got the word "a" in them.

Next of all, let's find term 2, which might be "and”, let's say. And the word "and" appears three times starting at document position number 3. And there's three of them, so it's three, four, and five. And it ends at number six.

So, three, four, and five values here, three is document 1, four is document 2, and a five is document 3. So, it appears in documents 1, 2, and 3. So, we know that "a" appears in document 1 and 2, and "and" appears in documents 1, 2, and 3 and so on.

I can run on term three, which only occurs once, let's say aardvark. And this is position number 6. And its document number 2. So, "aardvark" appears in document number 2 and so on.

So, a dictionary is a list of terms including normalised keywords or stems. So, you might have "aardvark" but not "aardvarks" because "aardvarks" is included in "aardvark" and the frequency with which the term occurs. So, "aardvark" only occurred once, whereas "and" occurred three times and a pointer into this inverted file and then the access to the dictionary. If you want to look up a word like "aardvark" in the dictionary, you just use some standard file access method like a hashing algorithm.

So, the inverted files got for each entrant a dictionary a list of pointers into a data file identifying those objects. And the inverted fire may also contain other information like positional information, where in the document it is. In this example so far it hasn't, but we've got it. And the term frequency, how many times it occurs in a document, that is potentially useful because if there's more frequent occurrence, it's perhaps more important or perhaps less important.

So, let's try a query like A or B and C. Let's say it's a cold day out and I want to make some pudding. And I want jam pudding or treacle pudding. So, I want jam or treacle and pudding.

So, that's my query. I want to find all the documents which have got jam or treacle and pudding in them because I want to make a jam pudding or a treacle pudding.

And what you have to do is...well, the algorithm first of all is this rather complicated thing with open brackets, A or B, closed brackets, and C. It's too complicated. So, you have to convert it into what's called disjunctive normal form. That is x or y or z or x or y or z or other things.

And then you can basically try matching each of these one by one until you found what you're looking for. An A or B and C amounts to A and C or B and C or possibly A and B and C. Those are the three options that are matching. That was 101, or 011, or 111 in binary terms.

And then, the T... then we have a list of documents. And the first list is the ones containing "a", and the second list is lines containing B. And the third list is the one containing C.

So, in order to find a match, you basically have to try this disjunctive normal form against this list of three lists. So, the very first one, 101, is matches...it's in document 1. And it's not in document 2. Well, 101 matches it. It is found.

So, the A is in document 1, and C is in document 1. That's good. So, then we throw away document 1 or pop it off a list.

And the next thing is document 2 has a profile 011. And then we pop off document 2. And document 3 has a profile 110. And we pop off document 3, and document four has a profile 101. And we pop that off, document four.

And we see document 5 has a profile of 010 and so on. Document 6 isn't...has also got profile 010. Document 7 is 101. 8 is 110. Document 9 is 001. Document 10 is 100. Document 11 is 001. And document 12 is 011. So, OK, for each document, we have a straightforward profile which corresponds

to whether or not A, B, and C occur there. And then we basically see, does it match 101 or 011 or 111?

Having done all that, we see those are the only actual documents. Document 1 matches 101. Document 2 matches 011. Document four matches 101. And document 12 matches 011.

So, we've found four documents which have A or B and C. And it was documents 1 and 4 have got jam pudding. And document 2 and 12 have got treacle pudding. So, there's our answer.

And we can report the number of hits to the user. We found four documents, and we don't even have to actually retrieve the documents. This is only in terms of matching index terms. Then the user can say, yes, I like these and then go and get the document, so document 1 document 2, and so on, effectively URLs. And then if you want to click on them, then you can go and get the documents.

Of course, that's simply the word is in there or not in there. And that's not realistic. So, what's actually important is weighted weights. All the words have weights, and the words in the query have weights.

So, instead of saying A or B and C when you issue a query to Google, what we actually want is, let's say, I want jam. I'm not 0.5 as my weight for jam. I'm very even keen on treacle. And C, I must have pudding.

So, this is a vector for my query now. I'm saying, I probably want jam. I even more surely want treacle. And I must have pudding.

So, all the weights are in the range of 0 to 1. So, any other word has got a weight of 0 by default. We still have a list of each block for each of these key words, which documents they're in. Remember, this is the inverted file for each word.

For A, we have a list of documents it appears in. And we also have a weight of that word within the document. So, for example, if the word A jam, in document 1 it has a weight of 0.2, whereas in document 3 it has a weight of 0.6. Basically, it means jam is much more important in document 3 than it is in document 1.

Where do we get these things, these weights within the documents from? Well, it's something to do with a frequency of a document or the frequency of the word in the document compared to the frequency of your other words, something like TFIDF within a document. And I haven't got time now to go into more details. Have a look at the textbook, this Jurafsky and Martin textbook chapter on question answering does have a section on information retrieval and which will explain this a bit more.

So, we have to assume we have some sort of similarity function, which for any query and document comes up with a score by comparing these vectors. So, again, we can look. We can go down, do the same thing we did before, get a profile for each of the documents.

So, profile for document one is 0.20.4. And the similarity between the query, 0.5.71, and document 1, 0.204, using this function comes up with 0.85. So, what is this function? Well, it's cosine function that we saw earlier.

And we get the same...do the same calculation using cosine function for document 2. Again, compare the similarity of the weighted query vector 0.7 with the document vector, which is 0.6.4. And this time, we get an answer of 0.6. That means that document 2 is slightly better than document 1 for our query and so on.

The last thing about this, once we got these weights attached you can sort or rank the list according to these weights. So, we just said that the document 2 is better than document 1. So, it's presented to a user first.

We present the ranked list to the user in rank order. And you can say, I only want to have, let's say, the best two, in which case all the others that aren't in the best two are not offered. So, you just offer the URLs for the best two. And then you can say, if you want, you can get the next.

So, far, we haven't actually retrieved the documents. All that's stored in the inverted file is for each word a list of document URLs and the weights of the words within each of those documents. So, we don't actually have to start downloading these things.

Furthermore, you can do something a bit more complicated now. If you want to say not just, I want A and B, but I want A within three words of B or I want A before B, if you carry information in the listing in the file...in the index, sorry, about where the positions of the words are, then you may be able to do things, for example, to search for "Venetian blind" rather than "blind Venetian”.

And in principle, this is possible. I won't go into details of how it's done. But this does become important. You want a query for "Venetian blind" to favour documents which actually have the phrase "Venetian blind" in it but still allow it to find "blind Venetian" as a sort of second best if you found enough "Venetian blind" documents.

So, we can see that the inverted file can be used for Booleans, but also weighted and also even these positional queries, and query processing can be completed without having to upload web pages. Before the query is launched, you have to do a lot of pre-processing to create this index, so Google has to, before you ever take a query from a user, set up this huge index of all of the documents in the world for all the vocabulary that appears in all of those documents and then for each vocabulary item which document appears in and where the words are within that. So, it's got a huge amount of pre-processing to do.

But once that's done, the query and query processing and matching is very fast. The number of hits is available straightaway. It can be expensive to update if the information objects change.

In other words, if web pages change then you have to essentially recompute the entire index all over again. And unfortunately, web pages do change. So, Google has to keep maintaining its index.

Google is very demanding in storage requirements. The dictionary plus the inverted file can take up about the same size as the original data. So, you have to store the contents of the web pages and you also have to, because you need the [INAUDIBLE] of the web pages, to extract this dictionary in inverted file. But you have to store that too.

So, an inverted file index is a standard system for information retrieval. Relational databases aren't very suitable for doing this sort of thing. Standard SQL is not very good at expressing queries, find all the records which have got several keywords in them. That doesn't work very well in SQL.

The inverted file structure is much more efficient, and it can also store frequencies or weights in the dictionary to allow you to order the results rather than just saying, I've got four matches. You can get a score for each of those matches, and then you can put them into some sort of ranked order. And this allows you to find quite subtle things like the difference between "knowledge management" and "management knowledge”. If you query both of those things to Google, you may find similar documents but in a different order.

And remember, Google has to find documents very quickly. I say Google. So, do the other search engines too. It just happens that Google is probably the most famous at the moment.

This is done very efficiently. And so, the cache, basically the Google robot goes out on the web all the time finding new web pages, storing a copy of a document and also a cut-down version of a document. When I say document, I mean any web page. That counts as a document.

It has to keep the words in the document and a stored, sorted list of the words appearing in a document with links back to the full document. So, that's done via this dictionary and inverted file. So, there it is again.

So, we've said that information retrieval is quite different from database management systems. It's different in a number of ways. It's different in the underlying algorithms that are used, but it's also different in what is expected of it.

So, for a database management system, you expect to issue an SQL query and get exactly what you asked for. With information retrieval, you type some query into Google, and you get a partial or best match, and you get a rank ordering of matches. It may be some of the lower-down matches are very partial and not matching completed at all.

The inference mechanism in database management systems is deduction. Whereas information retrieval, it is induction. If you're a philosopher, this can be important. Probably, it's not that important for computer scientists.

But what is important more is that the underlying mathematical model is deterministic for database management systems. That is, if you issue a query, you should be able to predict exactly what you get back. And the same query you should give you back exactly the same information every time. Whereas information retrieval is more probabilistic or that the weights attached involve random probabilities to some extent, which means you may not get back the same query, the same responses, or the same ordered responses the next time you issue the query.

The data in database management systems is structured into records, and the records are structured into fields and subfields. So, it's very structured. Whereas information retrieval, you basically have a set of documents, and we don't know anything about the internal structure of a document except that they contain text.

And the query language of SQL is pretty artificial. I find it quite hard to think about. Maybe after you've learned it, maybe you can get used to it.

For information retrieval systems, Google lets you say in sort of natural English. I say question mark natural because you have to think of some key words and then type them in. You can issue a question to Google like, who is the queen of England? And then Google will try to find an answer to it, will treat that as a question rather than a query. So, it's sort of quasi-natural.

The query specification, as I said, has to be exact and complete for SQL. Whereas for information retrieval, you can have a guess as to some keywords. And it may not be the complete, best answer. But you may find what you want even with just an incomplete query.

And that's because you don't necessarily want exact matches. The items that you want from an SQL query are exactly matching the query. Whereas for information retrieval, what you want isn't actually a match to the keywords. It's got to be something relevant to what's in your mind.

So, if I'm looking for...let's say I want to find out the Russian word for "table”, I might put into Google "Russian table”. And it comes up with answers. And the answer is correct if it gives me the Russian word for the word "table”.

Whereas my friend might be looking for a table of Russian-to-English translations. And then he puts in "Russian table" into his query engine, and he gets back a translation table of Russian words and of English equivalents and then he's happy. So, relevance is dependent on what you actually want rather than what the exact match is.

The error response is also different. If you put in an SQL query and it's not quite right, then you'll get something completely wrong. So, it's sensitive to errors.

Whereas information retrieval, if you type in a query and it's not quite right, you'll still get more or less a match. So, it's relatively insensitive to errors. So, Google is more error resistant or error insensitive.

Information retrieval was first of all for libraries, for bibliographic systems. So, they still refer to documents as what you trying to look for. It's not just web pages.

And the idea is you'll have your representation of a document as a set of descriptors or index terms or words in the document. So, you are searching for a document in the space of index terms or words. So, we have to have a formal language for formulating queries and a method for matching queries with document descriptors or keywords.

This is the general architecture. The user types in a query of keywords. The query matching looks at the object base. That is the database of the base, the record of all the web pages, all the documents. And it finds ones that match and returns hits.

Now, sometimes the user may say, I didn't like these. But I would more like this one. That's some feedback. And there may be a learning component which lets you give better hits and more of what you want and less of what you don't want. We look at query broadening a bit later on.

So, let's look at some worked examples of Google in a very simple world. Let's say we have a very simple set of documents D and a very list...small vocab T of terms that you're allowed to. So, for any particular document, the document is represented as a list of the words that are in that document.

And so, let's just say, for example, there in our very simple world wide web we've only got three documents, D1, D2, and D3. And the only words of interest in any of these documents...let's ignore all the function words. The only interesting words are "pudding”, "jam”, "traffic”, "lane”, and "treacle”. So, pudding, jam, and treacle, you may remember from my earlier example. It's cold outside, and I want a jam pudding or a treacle pudding.

But the documents can also contain "traffic" and "lane”. So, here's these three documents. The first document 11000 represents the document. That means it contains pudding and jam and does not contain traffic, lane, and treacle.

The second document contains traffic and lane, but not the other words. And the third document has got pudding, jam, traffic, and lane, but not treacle. So, what are these documents?

Well, let's just imagine the first document is a recipe for jam pudding. That's why it's got jam pudding in it. The second document is a Department of Transport report on traffic lanes. That's why it got traffic lanes in it but not the other words.

What about a document which has got pudding and jam and traffic and lane? Well, this is a radio item on a traffic jam in Pudding Lane. Pudding Lane is as a road in London where the Great Fire of London started. So, I've actually been to Pudding Lane to have a look, and there's lots of traffic there all the time because it's in London. And there could well be a traffic jam.

Notice that this simple representation does not capture any bigrams. It only captures unigrams. So, it doesn't capture the idea that jam should go with pudding and not with traffic.

So, those are the three documents. Now we want to find our recipe for something to eat tonight. Let's say in the Boolean model the query is something like I want jam or treacle and pudding and not lane and not traffic. And I've got to convert this into this disjunctive normal form.

Our vocabulary is pudding, jam, traffic, lane, and treacle. That means I want jam. I want pudding and jam, but not the other ones. Or I want pudding and treacle, but not the other ones. Or possibly, I want pudding and jam but not the other ones, so 11001 represents pudding, jam, no traffic, no lane, and yes treacle.

My original query was I want jam or treacle and pudding and not lane and not traffic. But we can't have these AND’s. We've got to really express it in terms of OR’s something or something or something. So, then I can quickly match against the internal structure.

So, to match a document with a query, you have a similarity function. And since this is the Boolean world, the similarity function is yes, 1. The score is 1 if a document equals 1 of these query vectors, if it's either 11000, or 10001, or 11001. Otherwise, it's 0...a very simple similarity function which is either one or 0, depending on whether or not we found an exact match or not.

So, let's take this in a Venn diagram to show the possible combinations. The first document is 11000. Does it match this or this or this? Well, it matches the first thing, 110...it matches the first component of the query. Therefore, document 1 is a match. The second one is 00110. It doesn't match that. It doesn't match that. It doesn't match that. Therefore, that's not a match. The third document 1111...sorry, 11110, it doesn't match that. It doesn't match that. And it doesn't match that.

Therefore, that's not a match. Therefore, we found the first document is the only thing that...it's a jam pudding recipe. And that will do for us. So, we found by collecting the results the answer is that document 1, 11000, which is the jam pudding recipe.

So, I'm happy. I have a jam pudding recipe to cook myself a nice pudding tonight. And if you like Venn diagrams, there's the intersection.

That's OK if you only have 1's and 0's. But in real life or even...this is a very simple world. But we're still going to insist on having weights. We want vectors with numbers and weights in the range of 1 to 0. It's not by binary value.

So, the query also has to be as a vector. Remember, there are five terms. And the first term, remember, was "pudding”. I must have pudding, so pudding gets a weight of 1.

I'm quite keen on jams. So, that has a weight of 0.6. I don't want traffic or lanes, so they get 0's. I like treacle pudding even more than jam pudding, so that gets a weight of 0.8. So, that's my query. 1, 0.6, 0, 0, 0.8 is what I want.

And the documents also have for each of the words a weight between the 1 and 0. There's a similarity function. This time it's this if you remember from before. But this is the cosine coefficient.

And the cosine coefficient, it has only two terms involved. Then if we have, for example, document one has a lot of T1 and a little bit of T2 and document 2...the query, sorry...the query has a lot of T2 and a little bit of T1, then this angle there, the angle is equivalent to the cosine of the angle is equivalent to this equation. So, if you can calculate this, then we have a cosine coefficient.

And the point is given a document and a query which overlaps, if the weights of the words are in the same proportions for the query and the document, then that angle is 0. And it turns out that the cosine of 0 is 1, so that's a perfect match. On the other hand, if a document has only got T1 in it and the query has only got T2 in it, then that angle is 90 degrees.

And this calculation works out. There's cosine of 90, which is 0. There's no overlap between the document and the query.

So, that's why we use this cosine function. So, let's go back to our query. I must have pudding. I'm quite keen on jam, and I definitely don't want traffic and lane. And I'm very keen on treacle.

Well, the first recipe, which was the jam pudding recipe, it's got lots of pudding and lots of jam in it. It doesn't have any treacle. It doesn't have any lane. It does have a minor mention of treacle.

So, it doesn't have any pudding, doesn't have lane, have a minimum of treacle. So, this jam pudding recipe maybe in it somewhere it says, get your jam and get your pudding. But if you haven't got any jam and you could get some treacle instead of jam, then use that whenever we say jam.

And then you get your jam, and then you get the pudding recipe. And you mix it all up together, and there's your jam pudding. That's maybe what it says.

So, we do the calculations. That's 0.8 times 1 plus 0.8 times 0.6 plus 0 times 0 plus 0 times 0 plus 0.2 times 0.8. And that gives us the sum of these multiplications. That's 1.44.

And then the sum of the squares is 0.8 squared plus 0.8 squared plus 0 squared plus 0 squared plus 0.2 squared. That's 1.32. And the sum of the query weight squared is 1 squared plus 0.6 squared plus 0 squared plus 0 squared plus [INAUDIBLE] squared plus 2. And then you put these terms into this equation. That's 1.44 divided by the square root of 1.32 times 2.

And the overall answer is 0.89. If you want to, you can do this calculation on the back of envelope yourself. Or you could just take it for granted that I've done the right maths here. So, that's the answer. The jam pudding recipe gets a score of 0.89.

Well, let's look at the next document. The next document, this is the report on traffic lanes. And there's no jam. So, there's no pudding, there's no jam, and there's no treacle. That makes this calculation fairly straightforward.

It's 0 times 1 plus 0 times 0.6 plus 0.9 times 0 plus 0.8 times 0 plus 0 times 0.8. That's just 0. 0 divided by some other stuff which we can calculate very quickly. But actually it's 0 divided by the square root of blah, blah, blah, which is still 0.

So, this is very efficient. If you get 0 on top, then you don't have to calculate all this other stuff. It's just 0. So, this Department of Transport report, even though we're now using weights, it still has a 0 score.

What about the third one, this third one, which was a radio report on traffic jams in Pudding Lane? Remember, we don't have any bigrams. We simply have the unigrams. And there is a unique...there is some occurrence of pudding and some occurrence of jam because there's lots of traffic jam. So, jam appears quite a lot in here.

In other words, the 0.6 times 1 plus 0.9 times 0.6 plus 1 times 0 plus 0.6 times 0 plus 0 times 0.8. At least the first two terms, pudding and jam, do have some significance here. So, it is 1.14 divided by 0.6 squared plus 0.9 squared plus 1 squared plus 0.6 squared plus 0 squared. That's 2.53. And the square of the query terms is 1 squared plus 0.6 squared plus 0 squared plus 0 squared plus 0.8 squared, which is 2.

So, this gets the score of 1.14 divided by the square root of 2.53 times 2. And that gives you a score of 0.51. In other words, the trade your traffic jam, radio traffic report of traffic jams in Pudding Lane, gets does get a non-zero score if I'm searching for pudding and jam and treacle because there are jam and pudding in this document.

So, now we collect the results. And I'm just saying we found two hits. We have a similarity score for each of these, which lets us rank them the best hit is the jam pudding recipe. The second-best fit is a traffic report about traffic jams in Pudding Lane.

So, let's have a look. The Boolean model is much simpler. There's a precise semantics. So, it's faster. But actually, it's not that useful because it doesn't rank the results. We've eliminated all but the one perfect match. Everything else is thrown out.

Whereas the Boolean model, it does work for bibliographic systems. So, if you go to Leeds University library and search for books written by Eric Atwell, for example, then you'll find exact matches, and you won't find probabilistic matches. It doesn't really work for very large web resources. And furthermore, users find Boolean queries hard to formulate. If you've got to put AND's and OR's in, it gets difficult.

There are fuzzy set models and extended Boolean models, but they don't really work too well either. The vector model is still quite simple. And its results show that it leads to fairly good results because you can have partial matching. So, some documents which aren't perfect still get a score and that then lets you rank the output.

This is why it's popular with search engines. It's still unrealistic in some of its assumptions. It assumes term independence. So, we're still looking for traffic and jam without realising a traffic jam can go together, or we're looking for pudding and jam but not realising that traffic jam isn't an appropriate pudding.

Phrases and qualifications are not really taken care of properly. Again, there are generalised vector space models which can take into account bigrams. But they get more complicated.

We haven't really talked about where these index terms come from. In our very simple model, I just said the only words we're interested in are pudding, jam, traffic, lane, and treacle. And we ignore the other ones. But actually, in real life Google has to allow for all the words in the documents as being possible search terms, and that can get to be a very large number of words.

Also, where do we get these weights from? Well, I just said it's something to do with the importance of a document, the importance of the words within the documents. Well, Google has got some quite sophisticated mathematics for working out weights of words within documents and all these other weights too.

How well do these systems actually work for practical applications? Well, that's the difference between Google and Yahoo and Bing. They have very slightly different weighting formulae. And they try them out on different populations.

Google may try. Let's try this different weighting. We'll try it out on California first. And if the people of California seem to like it, then we can use it, extend it to the rest of the United States and then the rest of the world. That's basically what they do. They keep doing experiments by trying out variations of things.

We'd also like to integrate information retrieval into more traditional database queries, question answering systems, and so on. So, nowadays if you type into Google some queries, it will find matching documents. You can also type into Google a proper question like, who is the President of the United States? And Google will recognise this as a question, and it will look into Wikipedia or other data sources to find answers.

We still haven't dealt with...so far, we've assumed a search term is just a unitary term. But what happens if I want to search for football and I also want to find web pages that have the word soccer in them? Soccer is the American word for football.

At the moment, the model doesn't really do that. It has a separate index term for football and soccer. So, we'd like to be able to search for one and find both.

In America, a faucet is what we call in Britain a tap. If you want to search for one of these and find results of both, those are synonyms. There's also ambiguity the other way.

So, tap in English can mean what the Americans call a faucet, the thing that water comes out of. But it can also be a bang or a sort of dancing where you tap on the floor a lot. That's quite different from tap. A word like tap can have different senses.

Of the word L-E-A-D can be a metal, lead, or it can be a lead for taking a dog for a walk. So, an individual keyword can have several senses. Or alternatively, a particular sense can have more than one keyword. And you'd like to be able to somehow link those.

Also, the context that the word appears in may be important. So, for example, if I'm looking for articles to do with football, they may mention names of players and clubs and things but not actually the word football because it's taken for granted that the document is about football. So, there's no need to use the word football. And also, within football documents, the word goal would probably mean putting a ball into the back of a net. Whereas in a management document, the word goal will be the objective or aim that you're trying to reach.

And as I said before, we have this problem of multi-word terms. The Venetian blind is not the same thing as a blind Venetian. A blind Venetian is someone from Venice who can't see, whereas a Venetian blind is a particular sort of window covering.

So, how do we evaluate these information retrieval systems? Why is Google better than Yahoo or Bing? Well, one is that there's all sorts of metrics, not straightforwardly accuracy at all, but other things like effort. How difficult it is for users to formulate a query which gets what they want? Or time, how fast is Google in producing its results compared to Yahoo, let's say?

The other thing is presentation of the output. So, Google prides itself in presenting its results very straightforwardly, very clearly without much graphics. Whereas Bing, for example, has decided it's useful, that users actually like to have some graphics included. Coverage of the collection, how many web pages are actually included of all the web? So, Google has prided itself on trying to find every web page out there.

Then we have accuracy, recall, precision. Those are metrics for measuring the results that you're given. And there's also things like user satisfaction. How satisfied am I with the results?

Assuming that what you've got is not perfect, how do we get better? Query broadening is essentially trying to find more of what you want and less of what you don't want. A user is typically unaware of what is in the web pages in the collection or how to formulate the query which best finds what you want. So, query broadening is an approach which takes the query that the user gave...a naive query, not necessarily the best possible query...and tries to find a better one which gives you more of what you want by either finding new index terms that the user hadn't used or adjusting the term weight to get more of what you want.

So, how does the system do this? Well, there's two sort of general approaches. One is to ask the user for some feedback. Tell me of the results I've given you, click the ones you like and don't click the ones you don't like. Another method is to offer the user a thesaurus or a term bank to broaden the query themselves.

So, let's look at user feedback versus relevance feedback. Let's say the system gives you a set of all the hits. And of these, the user decides this one and this one are relevant and the others are not relevant.

Then the idea is, given the query, you replace it with a new query. And the new query Q dash is some constant alpha times the original query, so assuming the original query was reasonably good. So, we'll take that into account.

And then you also add on some constant beta of the sum of all the good hits, the relevant hits. Assuming that the query is a vector, and the document is also a vector you basically take a fraction of the original query, and you add on a sum of all of...sum together all the good documents and take some fraction of those. And you can take away all the bad documents added together by sum weight of that. So, this essentially comes up with a new query which moves the query vector closer to the centroid of the relevant, retrieved document vectors and further away from the centroid of a non-relevant, retrieved documents. So, you kind of have a new query, which is like the old query but has got a bit more of the good hits and a bit less of the bad hits in it.

And the point of this, we expect that documents that are similar to one another in meaning to have similar index terms. So, the system creates a replacement query based on the initial query but adds index terms and weights that have been used to index the relevant documents, increasing their weights and also reducing the weights of terms found in non-relevant documents. So, how could this help?

Well, if the user uses the word jam and if some of the recipes that are actually relevant have the word jelly in them, then jelly turns out to be a new term that wasn't used in the original query. But a lot of documents have got jelly in them. Therefore, we start adding a small weight of jelly in. It happens that the American word jelly is equivalent to the British word jam. And now hits can use jelly as well as jam.

The other way around, if a user wants to have documents about lead and it gets documents related to dog walking because it has lead in them, so you've marked those as not relevant. And you downgrade all the other words, which appear in taking a dog for a walk. For example, dog and walk will be reduced. And therefore, other documents which have got lead in them, which also have dog and walk in them will be downgraded in future.

It may be that actually we want to set gamma to 0, ignoring non-relevant hits. So, we only have positive feedback. Because quite a lot of the hits may be non-relevant, that adds some sort of noise to it.

The basic feedback mechanism is the user is invited to tick the ones that they like. And then for those ones only, you add in some extra weights, or you increase the weights on queries, word terms, which appear in those documents. This feedback formula can be applied repeatedly. So, you give the users some more results and again say, which ones are relevant? Typically, you only do it once, but you can do it repeatedly.

This is particularly important for high-use systems. For example, we did have a PhD student who worked part-time in the British Library finding documents. His job was to do human information retrieval.

However, one drawback is it's not fully automatic. The user has to specify, I like this one and this one and I don't like those. Let's get a simple example going back to the documents.

Now we've got four documents rather than three. Our vocabulary is still the same. We want pudding, jam, traffic, lane, and treacle. The first document, which is a recipe for jam pudding, has got jam and pudding and a little bit of treacle in it. The second document on traffic lanes has got lots of traffic and lane, but no pudding or jam or treacle.

Then we have a new document we didn't have before. This is a recipe for treacle pudding. So, it's got pudding, but it doesn't have jam in it. But it has got a lot of treacle in it.

And then the fourth document is this radio item on traffic jam in Pudding Lane. And then if you do all the calculations, I mean, the query initially, my initial naive query, is I want pudding.

And I'm quite keen on jam, so I haven't even thought about treacle at all at this stage. So, I definitely want jam pudding. And there's my query, 1 for pudding, 0.6 for jam, and 0 for everything else.

If we do a calculation using the cosine coefficient thing, these are results. The first document scores 0.91. The second document, which is only about traffic lanes, gets a score of 0. The third document gets a non-zero score because there is pudding in it, even though it's treacle pudding. So, it gets a 0.6.

Then the fourth document has got lots of jam, but traffic jam, and lots of pudding, but Pudding Lane. And that said, it's got quite a high score. If you only want the best two documents, then we get the recipe for jam pudding is the best. And the radio item on a traffic jam in Pudding Lane is the second best.

And now we ask the user, which one is actually relevant? And the user says, the first one is relevant. But the second one is actually not relevant. We didn't want that.

Now we use this information of non-relevance in a clever way. What we do is we say we want more of the relevant document and less of a non-relevant document. So, we will return two documents.

So, the new query is setting alpha to 0.5 and beta to 0.5 and gamma, the negative feedback, to 0.2. We want 0.5 of the initial queries. The initial query was pudding and probably jam and 0 for everything else, so half of that.

And then I want to add on the good document, the one we thought was relevant. And the good document, that was the jam pudding recipe, which had 0.8 for pudding and 0.8 for jam. But it also had a mention of treacle. So, it says something like, if you haven't got any jam, then you could use treacle instead. So, it does mention treacle.

So, we get 0.5. We add on a half of this. And then we take away a small amount of a non-relevant document. That does have pudding and jam in it, but we're also taking away treacle...sorry, also

taking away traffic and lane. We're not taking away any treacle because there's no mention of treacle in this.

Now do this calculation. You end up with a new query. This new query is quite a lot for pudding, quite a lot for jam. It's got a negative value for traffic and lane. But now it has a very small positive value for treacle. So, treacle is also good because of this 0.5 of adding the good document.

So, now we've got a new query. And now we feed this new query into our cosine coefficient to come up with for each of these documents a score with a new query, with this new query we feed it in. And the new scores are 0.96 for document, 0 for the traffic lane thing, 0.86 for the treacle pudding recipe because it's got pudding and it's also got treacle in it. And the traffic jam in Pudding Lane is still there, but it has a lower score because you've got downgraded values for these things.

So, now if we ask for the top two hits, the top one is still the jam pudding recipe. But the second one is the recipe for treacle pudding. So, this giving feedback event gives us a new query which is better than the initial, naive query because it upgrades new terms that we haven't thought of, but which appeared in the good documents. So, the first document and the second one is also relevant. So, that's query broadening, giving user feedback.

Another method for query broadening is to use a thesaurus or ontology. A thesaurus is a list of words and other words which are related to it. It's a controlled vocabulary of terms or phrases in a particular topic.

It has classes of synonyms, words which mean more or less the same thing. And we also have a hierarchy defining broader terms, or hyponyms, and narrower terms like hypernyms. So, for example, Word Net or Roget Thesaurus is has got this.

And you could also have this in medical terminology. We have something called SNOMED, or the standard nomenclature for medicine. So, what we can do this is...so for example, let's say if we want

to replace words with documents and query words with synonyms from a controlled language, this can improve precision and recall.

So, for example, in a document, we have any old keywords. And we change the keywords into a standardised term using this thesaurus. And also, if you use a query, you can have any old words. And the user query is standardised to give us a normalised query. And the normalised query has a better chance of matching the normalised index terms in the content.

So, for example, let's say the document contains data processor, and the query is electronic computer. Well, the thesaurus says a data processor is computer sense one. And the query electronic computer is also computer sense one. Therefore, there is a match. If we replace the term in a document with a controlled language and replace the term in a query with the controlled language, then we have a narrower term, which gives us better scores, increases recall and precision.

So, that's another way of doing it. So, you can normalise the queries and terms in the query and terms in the document. And that potentially can be done automatically.

I think we'll stop there. Here's some questions for you to think about. You should know by the end of this why a traditional database is unsuited to retrieval of unstructured information. It's just too complicated. SQL doesn't let you ask for documents which contain A or B or C.

You should remember that you can't actually specify Boolean queries like A or B or C and not D. You have to convert it into disjunctive normal form. There is a way of doing that. But in principle, you don't really need to know how to. But you should have a go for simple examples.

The matching coefficient used for weighted similarity function always has a value between 0 and 1. And the similarity of A with itself should be 1. That's because the similarity function is equivalent to the cosine, and cosine for any angle is between 0 and 1. And the cosine of similarity between something and itself is equivalent to an angle of 0, and the cosine if the angle is 0 is 1.

So, you should have a clearer understanding of the difference between the vector model and the set theoretical model in terms of power of representation of documents and queries. The set theoretical Boolean model is much simpler. But it doesn't really allow ranked ordering of results, or it doesn't really allow for query broadening to improve the results.

So, that's information retrieval in a nutshell. We've looked at the difference between information retrieval and database querying through SQL. We've looked at the inverted file as a way of doing much more efficient matching in information retrieval we've compared the Boolean or set theoretical model against the weighted vector model. And the weighted vector model is much more like the thing that Google actually works with because it allows you to rank order the results.

And then once you've got a rank order, you can even give feedback about which results are good and give you a better query to give more of what you like and less of what you don't like. We've gone through some worked examples, so you can see some of the maths behind it. If you want to do more by looking in the textbook by Jurafsky and Martin, they've got more examples for you to play with.

You see, an evaluation, well, there's a number of different metrics. You can measure precision and recall of results. We can measure other things too.

And you could do query broadening to improve the matches. And I think I'll stop there. So, thank you very much for listening and goodbye.

[END]