# **Introduction and learning objectives**

So welcome to topic eight in which we're

going to talk about Information retrieval. So to get started, let's just have a quick

look at our course learning objectives and pick out the the three that

are relevant to this particular topic. So it's 2, 3 and 5, we're going to be

looking at language analysis techniques. For information retrieval, we're going to

be thinking about how to process text in order to make it retrievable. So that's going to be things like

stemming, tagging, parsing and analysis of text. And we'll be looking at various

NLP libraries for doing that. And we'll touch a little bit on

evaluation looking at applications of statistical techniques to

natural language analysis. So brief look at the topic

learning objectives. The three of those will understand

the information retrieval fundamentals. So we'll look at it from a sort of

conceptual and we'll conceptual point of view and look at the sort of mathematical

underpinnings of how to represent text. Then we'll think a little bit about

information retrieval data structures. And finally we'll talk about applying

information retrieval techniques and principles. And a little bit like the previous topic

we'll think about it from really in two halves. The first half will be largely

conceptual but with a practical edge. So we'll be looking at the concepts and

principles behind IR in the first part. And in the second part, we'll look in

a little bit more detail around some practical implementations of how to build

your own information retrieval systems. So there it is, let's get started.

# **Boolean retrieval**

So in this segment we're going to

introduce the topic of information retrieval. And we're going to look in

particular at a type of information retrieval that's actually was one of

the first really to become popular and mainstream, known as Boolean retrieval. But to get us started,

let's just step back a moment and think about information retrieval

from a sort of broader point of view. So let's just try and define it first. Information retrieval is essentially

trying to find documents, and documents can mean anything in this

context, scholarly papers, emails, personal records, products, anything. That satisfies an information need, and we'll talk a little bit about what we

mean by information need in a moment. And the other things that characterize

information retrieval is that we're dealing here with unstructured information

as opposed to database lookup, which of course would be

structured information. And we tend to be searching

in large collections, that's what makes the problem interesting

and challenging and different. These collections are normally of

the millions of documents or more, so that presents a challenge. And as I mentioned that the matching

that we're doing is not really sort of databases, this is not SQL

lookup statements and so on. And we're really trying to

deal with textual data, unstructured data in the form of strings,

and other types of representation. Now, most of us are familiar with

information retrieval from the point of view of web search, and that's almost a

cliché characteristic view of such as web search, well, actually, no, it isn't. There's lots of other more interesting,

I would argue, challenges in other use cases. What we call professional search is

essentially where search is carried out often by intermediaries or experts

dealing with multiple document databases, curated information. So legal researchers, healthcare

information professionals, librarians, and so on. Then we have enterprise search, we've all

had that moment where we need to find the holiday form or the maternity

leave form in a corporate intranet. And we've hit that brick wall, as so often

it happens because corporate intranets are normally so poorly maintained. That's what we refer to

as enterprise search and that's another very challenging

context for information retrieval,. Then the site search,

think about it, e-commerce. So many large companies that have thrived

through lockdown of through the COVID pandemic have done so because they've

provided very effective site search for products and services that we've

all relied on in our daily lives. So that's a challenging problem and there's some very good

solutions out there. And then of course,

finally, there's web search. The sort of prototypical example that

we think of when we would do a search on Bing or Google or

our favorite search engine. So I mentioned information needs,

typically they're task related to help us make decisions, settle arguments,

discover facts, but not exclusively so. Sometimes search is a subset of learning,

in this context, we're all trying to learn

about information retrieval. So, without being too meta about it, some

of the searching that we might do around this topic is associated perhaps more so

with learning than an explicit task. And then a searching for leisure purposes

which a lot of us do on sites such as Twitter or Instagram or

other social media, YouTube, and so on. So they're all valid use cases for

information retrieval, so let's not get too fixated

on the web search use case. So, again, stepping back for a moment,

there's been numerous models of information retrieval process

published over the years. They've helped inform people's thinking

about how to build effective information retrieval systems. Dating back to what was called the classic

model like you'll see here illustrated at the top where you've got essential

information needs on one side and documents on the other. And they're brought together

through a query and matched against document representations. And then we've got what's

called the standard model, which builds on that by more explicitly

modelling the role of the user. And representing the iteration that takes

place with the query refinement around a particular search engine and

a set of documents. But my own preferences for the third of these models is what we call

dynamic model or the berry-picking model. Popularized by Marcia Bates and this encapsulates two very important

insights about information retrieval. It's not linear, very often what we

find changes what we seek and very often the measure of success is not finding the

perfect document, whatever that might be. But the accumulation of

insights along the way, so that's why I think the berry-picking

model is particularly effective because it neatly encapsulates that non linearity

and the aggregation of insights. So let's think about a simple

information retrieval problem. For example, let's say we're trying

to find which of Shakespeare's plays contains the words good and

battle but not fool. So how might we do that? Well, we've got Shakespeare's plays

as text files so we could use grep, which is Unix's command for

doing string matching. And so we could grep for

the words good and battle and then we could remove the words with fool. That's a very primitive approach,

it could work but it's very slow, it's tricky to do negation. And it's tricky to do

things like proximity, like finding good within three

words of battle and so on. So how might we do better? Well, let's revisit the term document

matrices that we first introduced a couple of times actually in language

modeling also in text categorization. And we've got an example here of

Shakespeare's plays in the columns and various words in the rows of this table. And you can see what's called the term

incidence matrix, this is a binary matrix, so it's just showing, does this term

appear in this document or not? So each document is represented

by a vector of zero or one values which you've

got four columns here, but obviously there'll be many more for

all of Shakespeare's plays. So to perform that task of

finding all the documents, or all the plays that contain battle and

good but not fool, we could simply do a bitwise AND. So we could take the first row 1011 and

we could AND it with the second row, which is 1101. And then we could take the complement

because you want the negative of the third row. So instead of 1010, we've got 0101 and

we do a bitwise AND to those three, and we get 0001,

which of course matches Henry V. So we've found the answer to that

question that Henry V is the only one in this table that has battle and

good but not fool. But is this approach going to scale? As I mentioned in information retrieval

collections are typically much much bigger than this. So for example, even a relatively

modestly sized collection of 1 million documents,

each with say 1000 words, might give us a vocabulary of

half a million distinct terms. So that would give us a very sparse

matrix because it'd be a million by half a million,

which would be half a trillion values. So that's a lot, that's a big number and

that's a very sparse matrix given that only 100, sorry,

1000 million are actually nonzero and I'll let you figure out where

that figure comes from this. So it's a very sparse matrix, so is there a better way of

performing this computation? Well, it turns out there is, and

we introduce one of the key ideas in information retrieval, which is

referred to as the inverted index. An inverted index basically stores a list

of document IDs for each query term. So for example, for battle, we might

have document IDs, 1,3,4,7, and so on up to 39,

although they're Shakespeare's plays. For good,

we might have 1,2,4,9 on up to 39 and for fool we might have 1,3, 8, and 13. And these posting lists,

as they're called, are variable sizes because each term can

occur in a variable number of documents. And they tend to be sorted by document ID

to enable efficient computation as we'll see in a moment. So the entire set of terms

is known as our vocabulary. And then typically what we would do in

our information retrieval pipeline, we do some tokenization, which we've

already introduced in topic two. We do some normalization to conflate

variant forms so that U.K spelt with full stops also matches UK spelt without

full stops in a query or a document. We do some stemming and lemmatization to

accommodate all the lexical variations so the matching still takes place. And we might do some stop word removal,

which we've introduced earlier and we'll talk a little bit more about

as we go through this topic. Then to do our query processing, that's

where we introduce the Boolean operators. So for

Boolean retrieval we typically have AND which is the intersection or

which is the union and NOT which is a union operator which

negates whatever applies to you. So to perform our computation

as we discussed earlier, we would process the query battle and

good, we would look up the postings list for

battle, which we could say is length x. Look up the postings lists for

good, which we say is length y. And then we walk through those two

posting lists finding intersection and the time taken should be

of the order of (x +y). So for example here, we've got battle and

good we'd walk through the list, we'd find the match straightaway with

1 and we'd increment our pointer for the lower of the next

two values which is 2. Look for a match, no, we don't have one,

increment our pointer for the next highest value, which is 3. Do we have a 3 in the second list? No, we don't, so we'd increment

the pointer, we'd pick up 4 in battle. Do we have a 4 in good? Yes, we do, so that's another match, and

so on we carry on walking through this list, which is why you see now

why the sort order is important. And we'd make a note of all

the matches between those two lists. So that's essentially introducing

a very elementary ideas, Boolean retrieval, the operators,

posting lists, and term document matrices in order to form elementary

information retrieval computation. So we'll stop there for now and

I'll leave you with a thought to pause and ponder of that's how we would do an AND

query. You might want to think about how

we might process an OR query. And then in the next video we'll start to

talk a little bit more about term writing in different representation schemes.

# **Boolean retrieval practice quiz**

### Question 1

What are the key ideas behind the dynamic model of the IR process?

* It is linear
* It is non-linear
* It recognises the accumulation of insights along the way
* It aims to find the perfect matching document

### Question 2

What is the maximum number of non-zero entries in the term-document matrix for a collection of 1 million documents of 100 words each, with 50,000 distinct terms?

* 50,000,000,000
* 5,000,000
* 100,000,000
* 10,000

# **Ranked retrieval + term weighting**

In this segment, we're going to continue

our introduction to information retrieval and

we're going to look at a slightly different paradigms, than the one that we discussed

in the previous video. As you recall earlier, we introduced the topic

in information retrieval, and we talked about what was referred to as

Boolean retrieval, which is essentially

a set-based approach relying on Boolean

operators such as AND, and, OR and NOT, and so on. That often leads to

complex search strings, such as the one

that you see here, which is actually a search

string that's used to locate the profiles

for Java developers, and this is the search string

that a recruiter might use. If you've ever had that moment, a recruiter rings

you up and says, "I found your CV, I think

you're perfect for this role." This is how they find you, three strings like this. Boolean retrieval offers

very high degree of control. You can articulate

very precise queries using this formalism,

and it's transparent. You can see essentially why

documents were retrieved. You can trace it right back

to a particular keyword or a particular expression within that Boolean search string. It also facilitates

reproducibility, which is very important

in scientific research. Essentially, the idea is that results should be

objectively provable, and one of the key ideas behind that is

reproducibility of experimentation and

reproducibility of the methods that

gather that evidence. It's very important in healthcare information and

legal research too and in other areas that a search is

demonstrably reproducible. In other words, it

doesn't produce different results

on one day compared to a previous day or

different results for one person compared

to another person. Boolean retrieval

certainly facilitates that element of reproducibility. But since the web, things have changed

a little bit, search has become much

more commoditized and democratized.

Everybody does it. It's woven into the fabric

of our everyday lives, and most people rightly

or wrongly don't really want to learn

Boolean search. More importantly,

they want to have their results ordered

by some sole priority. On the web, we

tend to see what's called satisficing behavior. People don't need to find

every single instance of a document that matches

their information need, they just want the first

one that's good enough. That means that the results

need to be ordered, and very often people

don't scroll beyond the first page or perhaps

two pages of results. That introduces a slightly

different dynamic. The question is, what

approach might we want to adopt to better

support that behavior? That's where we introduce

ranked retrieval. The idea here is

that we're returning matching documents

with some sort order. The query terms,

instead of being a structured search string or search strategy as it's known are essentially just keywords, unstructured keywords, and

then the system returns the top end results and they're

sorted by some criterion. Sometimes chronological

order, but more often than not

sorted by relevance. The question is, how do we

calculate this relevance? Let's go back to our

term-document matrix, and you can see an example here. Each document is

essentially a count vector, and it's of length V, where V is the vocabulary. You can see we've got

our four documents here, obviously many more and our

four document terms here. Obviously, they'll be many more, but you get the idea

that each column would be account

vector of length v. That leads us on to a representation scheme that we call the

bag of words model, and this is bag in the

mathematical sense i.e a set without duplicates. When we have a bag of words, we're essentially

taking all the content out of a document or a query, and we're ignoring word order. Sometimes that's okay,

sometimes it's not really okay. You can imagine

Venetian Blind is not the same as a

blind Venetian, but to a bag-of-words model. Sometimes to Google,

they are the same. Now we have to think, okay, we've got our bag

of words model, our simple representation, how are we going to

weight the terms in that bag of words? Well, we could start by

thinking about term frequency. If a word is mentioned

many times in a document, it's probably slightly more

important to that document, than a word that's

only mentioned once and certainly

more important, you would hope than a word

that's not mentioned at all. Term frequency is a measure

of aboutness for a document. The term frequency

tf is the number of times that document t

appears in document d. But a word that appears 100 times is usually not 100 times more important than a word that

only appears once. We tend to have

taken log to have a dampening effect

on this function. We tend to use

what's shown here in the equation that the weight for a term or a document is

essentially the log. If it occurs more

than zero times, we tend to take log at

one because obviously, then it goes once we take

the log, log 1 is zero. Otherwise, if it doesn't occur, we give it a term

frequency weight of zero. We tend to use one plus the

log of the term frequency. Then the score for a query

document pair is the sum of all the terms where the

query intersects with the document of all those

term frequency weights. Again, as I mentioned, we tend to add one to account for the fact

that log of one is zero. Then we also need to think about another complementary measure to include in the weighting, which is that rare terms, and I mean rare as in across

a document collection are more informative

than frequent terms. You can think about a comparison

here with stop words. Stop words are so ubiquitous, they occur almost

in every document. They don't tend to be meaning bearing in quite the same way. The same idea applies here is rather than having a

strict cutoff that says, these are all the words on a stop list, we'll ignore them. We can't try and

model this effect quantitatively by giving

a higher weight to those words that are rarer in the collection

because they have greater discriminatory power of adding to the retrievability

of a document. The document frequency

df is the number of documents that contain

term t. Now obviously, if that number is high, it means the time is fairly

commonplace and ubiquitous. Actually, it's an

inverse measure. It's inversely proportional to that term's informativeness. What we tend to do is take

the inverse and the log. We take IDF is the inverse

document frequency is the log of n over the document

frequency for that term t, where n is the number of

documents in the collection. It actually doesn't matter which base we take for the logs. But again, we take log 2 for the same reason that

we did with term frequency. That leads us on to the

term weighting scheme, which is very popular,

known as TF-IDF, which is the product

of the TF and IDF weights as shown here

with this formalism here. We're essentially

taking the weight for a term and a document is the term frequency multiplied by the inverse

document frequency. That's been very effective for many years as it accounts for frequency and informativeness, and it's built into a lot of the NLP platforms that

you'll be working with, certainly NLTK and scikit-learn and various other platforms. Then the final ranking

across all documents for a given query is the sum of

all those TF-IDF weights. We iterate through all

the terms that intersect for a given query with all the document pairs

that we consider, and the score for a particular

query document pair is the sum of all

those TF-IDF weights. That's it. In the next

couple of videos, we'll move on to a slightly

different representation, and we'll build on

what we've learned about TF-IDF weighting, and we'll see how we'll apply it to a full end-to-end

ranked retrieval model.

# **Ranked retrieval + term weighting practice quiz**

### Question 1

What are the key advantages of Boolean retrieval?

* Reproducibility
* Transparency
* Control
* Relevance ranking

### Question 2

TF.IDF weighting accounts for which of the following?

* Document frequency
* Document length
* Term frequency
* Word order

# **Vector space model**

In this segment, we're going to continue our discussion of information retrieval

concepts and principles and we're

going to move on to one of the most

fundamental models within information retrieval, which is known as the

vector space model. Let's think back to the previous video

when we were thinking about how to represent documents and we talked

about the bulk of words. Let's consider our

term document matrix, and we've seen an example here again using Shakespeare's plays. Essentially, each document

is a count vector. We've got the term

instances in each column. Therefore, each document is represented by a

vector of length V, where V is the vocabulary. Now if you think about it,

the terms are essentially dimensions here in vector space. That gives us a V

dimensional vector space, where V is the size

of the vocabulary. Typically, V will

be the order of many thousands for a document

collection of any size, and most of the time these vectors will be

sparse, they'll be zeros. For large vocabulary, most words will not appear

in a given document. That leads us on to thinking, how can we do this

matching process? It turns out that we can

actually think about queries also in the same V

dimensional space. What we want to do essentially

is we want to rank documents according to their

proximity to the query. So we need some similarity

metric. How might we do that? Well, one obvious method

seems to working in multidimensional vector space is to use something like

Euclidean distance. That's a reasonable initial

estimation to this, but there's a problem

in the sense that Euclidean distance

is going to be launched for vectors

of different lengths. If we consider a document d', which is actually

just document that appended to itself, clearly, document d' has

identical content to document d. But because

it's twice as long, Euclidean distance is still

going to be very large. So Euclidean distance is not

really the ideal measure. Instead, what we want

to think about is the angle between

document d and d'. If we look at the little

illustration here, you can see if we

got a query term shown as a vector in green, we can see that document d\_1, and this is just

obviously representing in two dimensions, but you could imagine

the same principle applies in

multidimensional space. You can see the query q is

actually closer to document d\_1 than it is to d\_2 because angle alpha is smaller

than angle theta. That's essentially what

we're trying to do with what's known as

the vector space model, representing queries

and documents in multidimensional

space and applying some similarity metric

between the query and each document to

try and rank them. What we want to do, as you can see here in the

illustration is we want to rank documents by the inverse of their angle

because obviously, a small angle is better. It turns out that the cosine of the query against

the document in this multidimensional space

is a suitable measure. But again, as we mentioned, we have to also think about length normalization because if queries are different length, documents are going to

be different lengths, query is going to be

different lengths, we need to normalize for that. So what we tend to do is divide each component by its

length using the L2 norm, where the L2 norm is the square root of the sum of the squares

as you can see there. That essentially makes each vector a unit

vector on the surface of a unit hypersphere because

we've normalized now so the radius is 1 and it's a hypersphere, x is

multidimensional. Each factor is then

defined only by its angle essentially or its direction rather

than its length. If you think back to

the thought experiment early with d' and d, where we appended d\_2 itself, because they are

now unit vectors on the surface unit hypersphere, the vectors

coincidentally would be identical because we've

accounted for the length. That gives us this formulation

here where the cosine of a given query and a given document pair is actually the sum of all the q\_i, d\_i terms which we'll define in a moment divided by

the two L2 norms, the L2 norm for the query and the L2

norm for the document. The q\_i and d\_i terms

rather than taking the raw counts out of

that document matrix, we can take the

TF.IDF weights which we'd introduced in

the previous video. So q\_i is the TF.IDF

weight we determined in the query and d\_i is the TF.IDF weight to

determine a document, bearing in mind of course you've got one query and

many documents, so you'll have many

query document pairs. Then essentially, for

length-normalized vectors, the cosine is simply

the dot-product. As you can see there,

the summation of all the q\_i d\_i terms. Then the retrieval process is essentially defined by

these five steps here. We represent the query as

a way to TF.IDF vector. We represent each document

as a weighted TF.IDF factor. We compute the cosine score for each page in the formulation

that we just introduced, we rank the documents

by the school, and then we return the

top end to the user. That essentially is what happens or certainly what did happen for a long time in

information retrieval. That was the central

paradigm for so much, and actually, you can

go a long way just using an approach just

as simple as this. Obviously, there's lots

of other NLP techniques that we could and should

lay on top of this, but essentially that's

what's happening in many, many search engine instances. In the next couple

of activities, you've got a chance

to practice this, to do some of the

calculations yourself, and to build your own

mini search engine to practice with

these techniques.

# **Vector space model practice quiz**

### Question 1

Why is Euclidean distance not a good metric for measuring vector proximity?

* It is sensitive to document length
* It is sensitive to document content
* It cannot be used with TF.IDF weights
* It only works with up to 3 dimensions

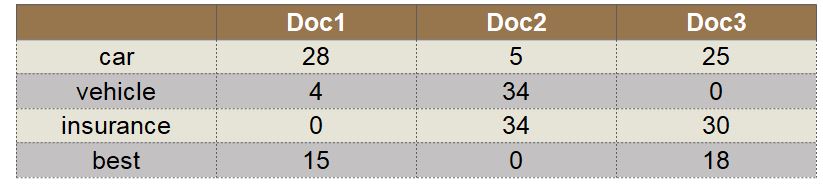
### Question 2

How do we normalise for length in calculating the cosine score?

* Take the dot product of the query and the document
* Divide each component by its length using the L1 norm
* Divide each component by its length using the L2 norm
* Take the inverse of the angle

# **Activity: Exploring IR models**

Consider the term following document matrix:



Compute the TF.IDF values for the terms in each document, given the following idf values: car=1.75, vehicle=2.18, insurance=1.72, best=1.60 Compute the TF.IDF values for the query ‘best car insurance’ Compute the ranking of each document for this query using the cosine measure.

# **Understanding IR models discussion prompt**

In the previous exercise you applied TF.IDF weighting to rank 3 documents against a given query. Write a summary for each of these bullet points and post it in the discussion forum. Why is the IDF value of a term always finite? What is the IDF value of a term that occurs in every document? Compare this with the use of stop words. What are the cosine measure scores of each document for the given query, and what is their final ranking? Once you’ve posted your comments in the forum, take a look at those of other learners and comment on the differences.

# **Representing documents**

Earlier on in this

topic we introduced Boolean retrieval and we look to document representations

and in particular, the bag of words model and

how we could use that to do matching in a ranked retrieval and also in a Boolean

retrieval sense. What we're going to do now

is we're going to take those ideas and have a look at how they might

work in practice, how we might represent a document as a bag of

words and how we might turn unstructured data into structures

that we can manipulate using data science approaches

and NLP approaches. To get us started off, we're going to import

something called CountVectorizer

from Scikit-learn. CountVectorizer does

exactly what it says. It takes counts from a term document matrix and

creates vectors from those. Then we're going to take

a tokenizer from NLTK. Obviously, you're familiar

with NLTK by now, and we'll use that tokenize

to split up our text. We'll do import. We'll import those

and we'll also define some helper functions

just to read the text and do some elementary text processing and

divide it up into sentences and we'll get the

sentences out of the text. For our little sample text, we'll use one of my blog

posts from a year or two ago. It's got seven sentences. Well, actually it's

quite longer than this, but there's just a sample here, has got seven sentences

here you see on each line. Now let's see how we might

create a count vectorizer, so we instantiate vectorizer using count vectorizer

and then like a lot of things in Scikit-learn, the paradigm is we

call fit transform, which basically instantiates

our vectorizer based on all the texts in the

sentences object and then we assign that to our

features matrix X. We can have a look at

x, see what's in it, and you'll see a very long list. To make sense of this, you need to interpret it as a tuple of a document

ID and a term ID, and then a count you

got document ID 0, you go term ID 6, and that occurs once. You can see actually if you

go right down the list, the document ID stop at six

because we've got no six, which is seven sentences I've just discussed

earlier and you see the term IDs go

up to about 119, but there might be another one, a couple of an ellipsis

here that was not seeing. It's at least 119 and you can see the term

counts here that this term ID 76 occurs three

times in document number 1. I reckon that's

probably the word it's that's occurring in

this second sentence. That's the features matrix which you see is

a sparse matrix, we can convert it to

a dense matrix using NumPy and we can see that that is a slightly

different representation, line by line representation

where essentially you've got each document number, then each document on each line. You've got a feature vector

of all the occurrences, the counts for all the term IDs. You've got 100 and, well, we thought it was

119 vocabulary size for each document

you can see here. Like document number 1, you've got term ID

number 0 does not occur. Next term occurs once the

next time occurs 0 and so on. This should be seven

of those stats are dense representation. You can see actually

if we do call shape, we can see that there are

indeed seven rows in that, but there's 121 elements in each so that gives

means essentially that answers our question

that the vocabulary is 121 insights and if we want to see what the

vocabulary looks like, we can get feature names. There we see that is the vocabulary represents

this feature name, the list of feature names

in our count vectorizer. What we could do now is, if we've got a new sentence, we could apply the transform

to it based on the fact that the fitting of that

transformers done on a slightly different vocabulary. This sentence, now is the

winter of our discontent may contain our

vocabulary terms. Again, we can have a look

at that and we can see that actually three of the

terms of matched, so presumably the other three

were not in the vocabulary. Say 57,76, and 104, were the term IDs that are present in both the new

sentence and count vectorizer and they're probably

the words is the end of and you can see the dense representation of

it there that the count, the three counts, non-zero counts in that dense

representation. That's essentially how to

use a count vectorizer to represent

unstructured texts and do elementary modeling on

it that we can use for information sharing lots of

other NLP related tasks. We can also use or apply

stop words to this by simply applying the premises

stop words equals and whatever language you want

will appeal pick English. We'll do a similar operation, instance count vectorizer, fit transforms sentences

and get the feature names. But this time with stop words applied and you can see

we've got another list, but all the stop words on are missing from this vocabulary, and I've recall the shape. We can see we're now down

to 76 we've had 121 before. But now we've got down to 76 because we've eliminated

all the stop words. Or if you don't

want to stop words, as we introduced in the

discussion earlier, stop words are fairly

ubiquitous so less informative, we can use the same concept

to document frequency. Remember when we talked about

inverse document frequency, we could apply maximum to this. If we say maximum document

frequency is three, which means eliminate

anything that occurs in more than

three documents. Again, fit the transforms, the sentences get feature names, look at the shape, and

we're now down to 119. There are two terms

that occur in more than three documents

and they've been eliminated from

this feature list. We could knock that down

to two try the same thing. Look at the shape,

and now's 114. We've lost seven terms

which presumably occur in three or more documents and we can knock it

down even to one. But just to

illustrate the point, and that's basically

saying there are 102 terms that occur once in one document and all the others

have been eliminated. There it is that's how to use a count vectorizer to produce a bag of

words model and you can use stop words

and you can use document frequencies

also to refine your features which

you can use in all information retrieval and natural language

processing tasks.

# **Representing documents practice quiz**

### Question 1

What do features represent in a CountVectorizer?

* Terms
* Lexical categories
* Stop words
* Documents

### Question 2

If you apply a document frequency cutoff of 1, how will this affect the number of features?

* It will decrease
* It will stay the same
* It will increase

# **Term weighting using TF.IDF**

In the previous video, we introduced the

CountVectorizer, and we saw how we could use

that to take a document, transform it into

a bag of words, and use that bag of

words for all sorts of NLP and information

retrieval related tasks. What we're going to do now

is we're going to continue with another concept that we introduced in the first

half of this topic, TF-IDF weights, as

you may recall. Although we talked about them in principle and in the

first half of this topic, we're now going to have a

look at them in practice. As we mentioned, TF-IDF

weights are a way to represent the aboutness of a document that

captures frequency. The term frequency

within a document, which obviously is a

measure of importance and aboutness but also accounts for scarcity i.e how rare certain terms are

within a collection. To get us started,

let's do our imports. Most of these are fairly

self-explanatory, but there's a couple here

that we should identify, which of course, instead of

importing CountVectorizer, as you saw earlier,

we're now importing TF-IDF vectorizer, we're also importing a

stemmer, a SnowballStemmer, and we've told you about

stemming in Topic 3, and we're also importing

our tokenizer, the same one as we introduced

in the previous video. Then we've got a few

helper functions which are now

particularly interesting, just reading in the

text and dividing up. Let me instantiate our stemmer

and we also instantiate a custom stopword list rather

than going with the default one that comes with

scikit-learn or an nltk, we're introducing our own, so we'll read to that end. We've got 476 words

on our stop list, and then we'll get

our sentences. Again, we'll take it

from my blog post, and you can see that there's seven sentences from it there. Now is the interesting

bit where we actually instantiate our TF-IDF vectorizer

with a few parameters. Notice how we're also

introducing minimum, maximum for the

document frequencies. Maximum features of 200,000, but although we

won't hit anywhere near that with just

this tiny sample. We're also introducing

our stopword list and also using IDF. As always, we fit it

to our sentences, and then we can create our TF-IDF matrix by

transforming our vectorizer, applying it to our sentences, so doing in two stages, where this is in one stage

in the previous video. That creates our TF-IDF

matrix which you'll recognize the structure of the tuple of the document ID

and the term ID. But this time, look at

the weights on the right, instead of being integers, they are real-valued counts, or rather real values, which represent the

TF-IDF weights. You can see a lot of them are similar probably because they have similar counts

and documents, and similar document

frequencies as well. Then you can see,

you've got your weight, that's the sparse vector

representations before, notice that the shape

is slightly different. We have seven rows but a greater number

of elements in each. You might want to pawn the why, if you are perfectly sharp-eyed, you may have seen

under the flag when we instantiated TF-IDF

vectorizer explains that. We can flip it to a

dense matrix format, as we saw in the previous video, line-by-line representation

which shows, as before, a line

for each document. But instead of there

being integer counts, we've now got real value

counts from TF-IDF weights. If you look at the shape,

it's still seven rows and 215 elements in each, which you can think of columns. If we get the feature names, here's the giveaway of why

this's a greater number, 215, I think we had 121, or something in the

previous count. It's because we instantiated TF-IDF vectorizer using N-grams. If you remember, in

language modeling, the unigrams are obviously the counts for each individual term,

but n-grams, bigrams, and trigrams gives us the pairs of words and

the triplets of words. You can see here, we've got unigrams, 10, and then we've got a

bigram, 10 topics, and then we've got

trigram 10 topics, theoretical, and so on. You can see that we've actually taken

into account where the TF-IDF vectorizer

has two things here. The counts are represented

by the real values TF-IDF instead of the role counts as this

one a CountVectorizer. Also, the features are

not just unigrams, there are bigrams, and trigrams as well. We could, I guess for

the sake of completion, try it with just unigrams. Then you can see

we've gone down 74, which as you'd

expect because it's unigrams minus the

stopword list, that's so much more

representation. There you go, there's

the seven documents with the TF-IDF values. It's still 74 and the feature names that we

just got the unigrams, so we're back to

where we were before, but it's also useful in

information retrieval to match not just on unigrams but also higher-order structures,

bigrams and trigrams. Again, these are the things

that you should experiment with for each information

retrieval task, different settings

will be appropriate, so it's good to know that these configurable

parameters are there for you to

experiment with. There it is, that's

the TF-IDF vectorizer, very useful, very accessible, very easy to use, and a very powerful tool for an NLP and information

retrieval.

# **Term weighting using TF.IDF practice quiz**

### Question 1

How is document frequency typically used in term weighting?

* As the log of the inverse
* As the inverse of the log
* As a raw count divided by N
* As a raw count

# **Semantic search**

In this segment, we're going

to pull together some of the threads that we've

seen earlier in this topic around representing

documents and using various weighting schemes. We're going to pull

them together in the form of how many

search engines. I should point out

that of course, real search engines are several orders of magnitude more complex than what

we're about to say. But it should give you a

little bit of an insight into how these things can be put together in a

relatively short space of time to create a

working search engine. What we're going to do

is we're going to search something called

the IMDB datasets, which is a database of movies. This is a set, what

you might call semi-structured data in

the sense that you've got some fields for

the rank of the movie, the title and genre. Then we've got this

description here, which is the free

text field that we'll use for searching

along with the title. But of course, bear in

mind that you could search other fields as well. But from a search

engine point of view, you can think of it as

searching a catalog and not catalog is a

database of movies. But more importantly, what

I'd like to introduce at this point is the idea

of expanding queries. Now this is a very

important concept, particularly within web search, because it solves what's called the vocabulary mismatch problem. What we mean by that is

when documents are drafted, they're often drafted using

a particular language, a particular genre, which is often the case if it's

a specialist document. When uses issue queries, they may not know that

jog and they may not know that particular language. They may use keywords

that don't actually match the terms in the document, even though the conceptual

level there is a match. How do we deal with that? One way of dealing with it is through something

called "Query term expansion," and there's

various methods to doing that. Some of the reading

materials associated with this topic talks

about those methods. But in this little exercise, we're going to use word2vec, which we've

introduced earlier in lexical semantics

and other topics. We're going to use the Gensim

library to instantiate a word2vec model and

we're going to use it for query term expansion to solve the vocabulary

mismatch problem. To get us started, we're here

in our Jupiter notebook. We'll do various import. We're going to import

something called whoosh, which is an open source library for building your

own search engine. We're going to also

import Gensim, which as we've met before, is another library

that allows us to do various approaches, dense factors, and

what to back and do all similarity matching

with language data. We're going to instantiate

a path to our model. We're going to use a

pre-trained model. You can see here model.bin. This comes from a collection or repository of freely

available pre-trained models. Here you can see

the vectors.npl.eu, there's plenty here

to choose from, but we're going to look at this one here that's

index number 40, which is trained on an

English language corpus using word2vec continuous

skipgram algorithm with a 100 dimensions and

a window size of 10. That's the model

we're going to use. We've got to set up a couple

of other little variables, the path to the dataset, which is in CSV form, and also a path to an index which is created

by the search engine, which is a little bit

like the postings lists, the inverted index that we talked about earlier

in the topic. Then we've got a definition

of the search engine, so at most, this is about assessing up the schema

and doing indexing. Not very interesting stuff, but the real interesting method is there at the

bottom query engine. I won't read through

every bit of the detail, but you can see that it parses the query and looks in the title

and the description, which are the two primary

free-text fields that we looked at when we looked inside

the CSV file a moment ago. Then we instantiate

our search engine, giving it the search engine path and the path to the dataset. Then in a moment

when that returns, we'll be able to load our model from the model

path that we had earlier. Gensim will then load up

model and we'll be able to call various methods of the Gensim a Gensim API in order to get

some of the words. Now, I'm going to

pause at this point and you may want to skip

this out or just edit. I may want to delete this

little bit while I'm waiting for the

prompt to return. There we go. Now we

can load our model, which calls the

Gensim method to load the model using the path

that we defined earlier. Again, this may take a couple of moments

so you might want to fast-forward this

bit or somebody might want to edit this bit out. Hopefully it'll return

before too long. That has now returned and as you can see

in the next cell, we have a method

called "Get similar words," which basically calls

the most similar method on the model and returns the top five matching words

for that search term. Now we can try it. Imagine

that we were searching for a movie and all we

knew is that it's had Galactic and it was

something like that. What we could do is we can instantiate the search term

for Galactic and we can call the "Get similar words"

method that we just defined [inaudible] of

all and the search term and that should return

in a moment with the matching terms

and then we can have a look at that and see

what it gives back to us. Again, that's just taken a

moment. That's returned. Then now we can look at the

other words that we got from the "Get similar words" method and you can see we've

got intergalactic, interstellar and so on. Now, what we can do is

we can call a method on our search engine called "Query engine," and we will

join our search term, "Galactic" with all

these other words in a large OR statements. This is exactly as we've seen before with boolean retrieval. Now normally of course,

this query expansion will be done inside

the engine and that'll be a more

elegant solution without an appropriate set of

parameters to control it. But for now, we're

just doing it in a rather simplistic

manner of calling the query engine method of

the search engine and just concatenating all these terms

into a large OR statement. We call that method

and that gives us the top six results. There's the large OR statement. In the method, we print out the first three and

the hits are number 1, 51, and 37. Remember, our query

was "Galactic" so we can see number 1 is

"Guardians of the Galaxy." That's quite a good match. We also have 51 and 37. We can see 37 is

interstellar again, another good match,

and 51 is Star Wars. You can see the semantic search

has actually worked here. It's taken a concept

of galactic, and it's returned three movies that all are

reasonably good match. Let's try another thing. Let's try a gigantic. We could again return that's, and that gives us

five different terms. We could call out on the search engine and

we get number 15, 151, and 881. Let's have a look at

what number 15 is. Number 15 is Colossal. Given that our

query was gigantic, that's again, a

pretty good match. I'll leave it as an exercise

for you to try out these yourself and look at whether the others will get

matches or not. Now remember, of course, you

need to configure the engine appropriately to search

the right fields. There's a whole industry around tuning search

engines and getting the relevance right and we're just scratching

the surface here. Just while we're here,

there's a couple of other useful methods that

Gensim provides that can give you all sorts of other interesting options

to control either a search engine or other

NLP applications. Here, we can define a set of words and we can

call a method called "Doesn't\_match," and

it will look among that set and find the

term that doesn't match. Courgette, mushroom, onion, and I'm sure the camera

is the odd one out. You can also call

a method called "Most\_similar\_to\_given," and

we give it a term fungus. It looks among those terms and it decides that mushroom

is the closest match. Again, that's pretty

reasonable answer to that question.

There you have it. You've seen woosh, a reasonably sophisticated

search engine library for building our own

elementary search engine. We've also augmented

that search engine with semantic search capability

to expand the queries. You can use this to refine it, to build on it, to create

your own search engine. Again, you can

combine the power of Gensim to do this query

expansion for you. There's a lot of

interesting possibilities using these basic methods.

# **Semantic search practice quiz**

### Question 1

What kinds of relationships are captured by gensim’s get\_similar\_words() method?

* Synonyms
* Related terms
* Translations

### Question 2

Which fields in a document index would normally be suitable for matching via semantic search methods?

* Numerical fields
* Boolean fields
* Text fields

# **Understanding Boolean search**

In the previous exercise you experimented with Boolean retrieval and query expansion. Consider the following questions, then post a link to your search strategy and your comments in the forum: Compare your search strategy with those of your colleagues. How do they differ? What can you learn from your colleagues’ attempts? Whose returns the most relevant results? To what extent did query expansion improve your search strategy? Once you’ve posted your comments in the forum, take a look at those of other learners and comment on the differences.

# **IR summary**

Okay, so

here we are at the end of topic eight, which is all About information retrieval. So let's take a moment to recap on

the things that we've studied and learned about. So if you may recall at the start,

we talked about the topic learning objectives, we said we'd understand

information retrieval fundamentals, and we did that in the first half of topic. Looking at some of

the approaches frameworks and ideas in information retrieval,

we surveyed analyze IR data structures and you had a chance to do a bit of that and

practice that. And we certainly did apply IR techniques

and principles, and we did so with a variety of exercises culminating

in building our own little search engine. So just to recap on

the things that we covered, we talked about information retrieval. We defined what we meant by information

retrieval talked about information needs. We differentiated information retrieval

from database lookup as focusing on unstructured information,

large collections, and dealing with essentially strings or

textual data. We looked at the various use cases for

information retrievals, not just web search, there's professional

search, enterprise search, site search. All these are very challenging topics. And there's lots of interesting

problems to be solved, and of course, I should point out that this is

one of my specialist subjects. So if you're interested in doing a final

project around information retrieval do reach out to me. I've always got interesting

projects available on that topic or you may have one of your own that

you're interested in pursuing. We also talked about the difference

between Boolean retrieval and writing retrieval. We looked at the strengths and weaknesses

of both approaches Boolean retrieval, obviously greater control transparency and

reproducibility. But those things aren't quite important

for web search, where you got more casual discretionary use, where people

don't want to learn Boolean and what their results ordered

in some meaningful way. And we talked about how we

might do rank retrieval. We talked about the notion of relevance, talked about using unstructured

queries as keywords and matching those using the vector space model, and

considered the term document matrices, and account vectors and

doing matching using the cosine measure. And of course, we also built our own little search

engine that did some query expansion. And we talked about how queries could

be represented in that space, and then we could rank documents according

to their proximity to the query. And we looked at various measures and said

we ruled out Euclidean distance and we finished up looking at the cosine measure,

and ranking documents by the cosine. So that's it really we've looked at

information retrieval from a theoretical point of view. We've looked at various

practical exercises as well. Pull those implementations together with

a mini search engine doing a little bit of query expansion. So it's a fascinating topic. It's one of my specialist subjects,

if you want to pursue it further, do reach out to me. But that's it for this topic,

and I hope you enjoyed it.