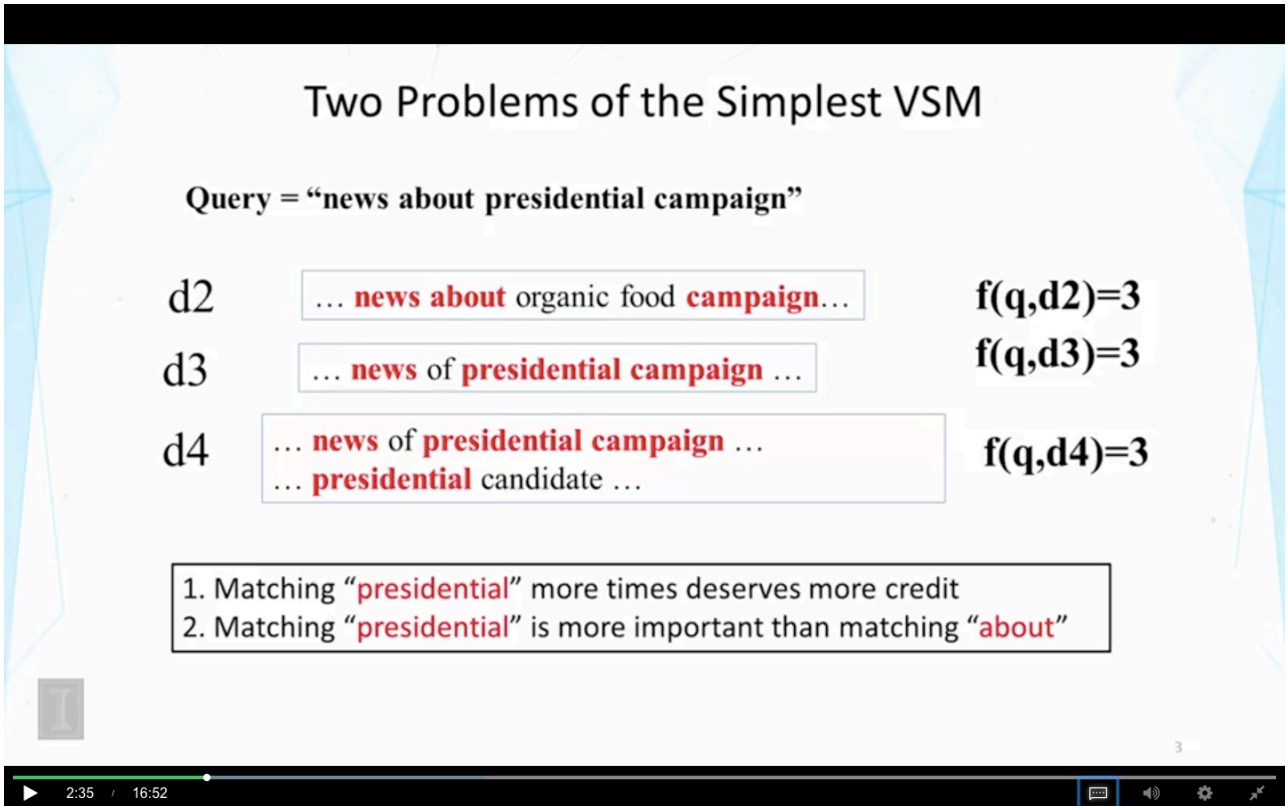
Week 2

Lecture 2.1 - (VSM - Improved instantiation):



These are the 2 problems of our simplest instantiation of VSM.

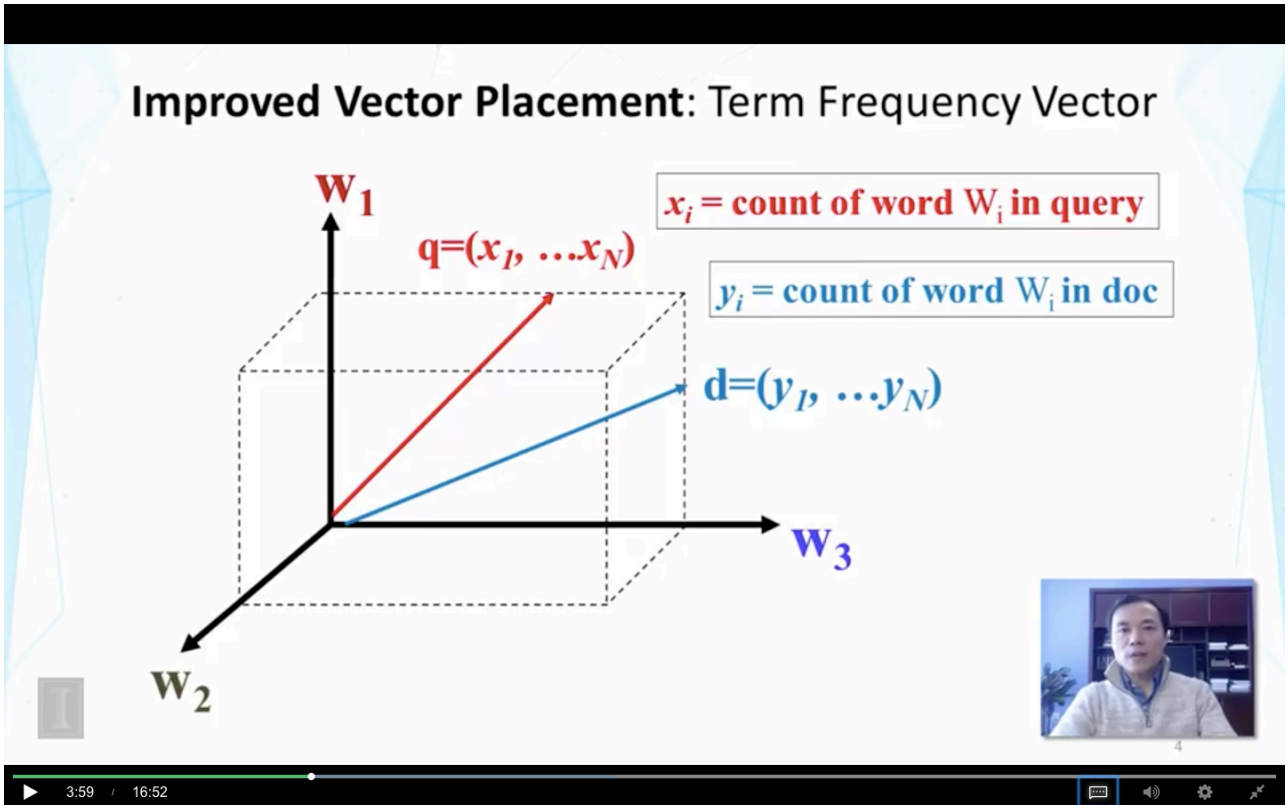
1. **Matching 'presidential' multiple times in a document to be given more preference**
2. **'presidential' vs 'about'**

Why do these problems exist?
These problems exist really from the assumptions that we have made in placing vectors in the vector space. So, we need to have different ways to instantiate vectors in the space.

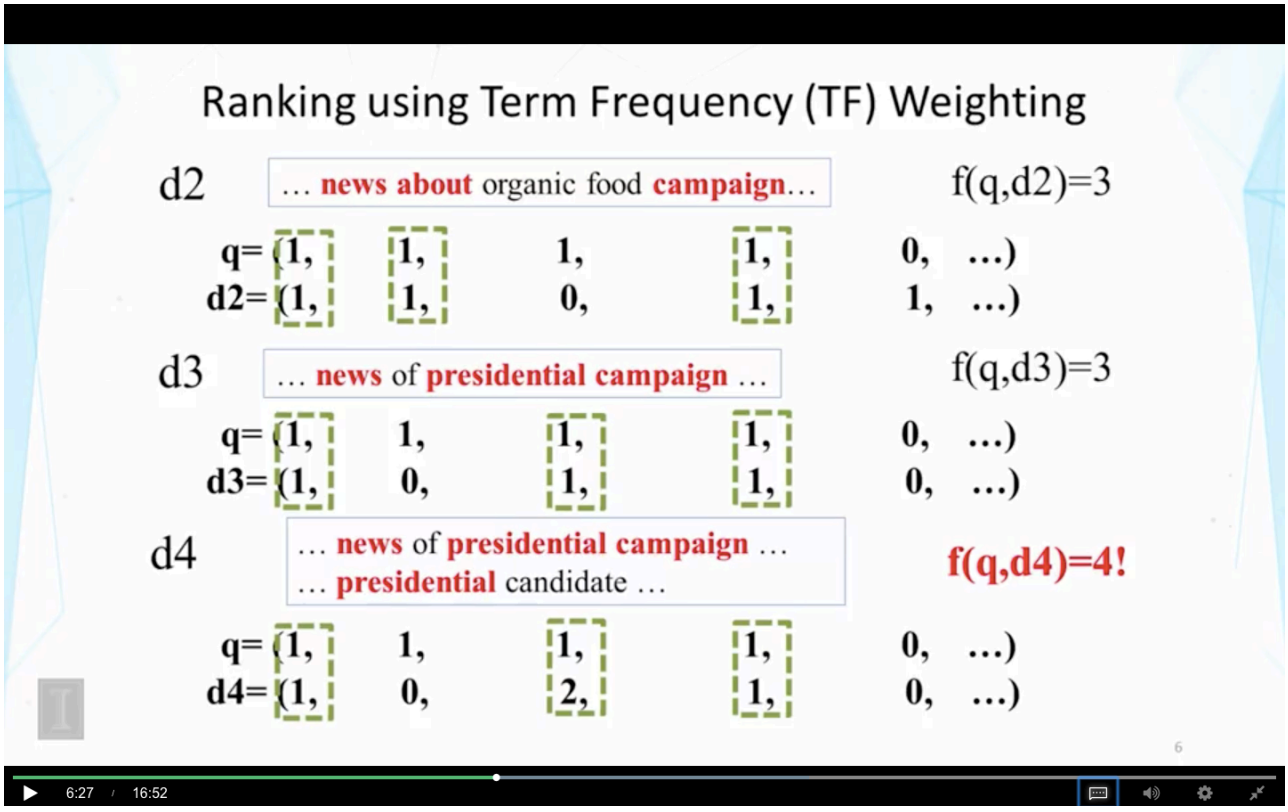
So, let's see the solutions to the problems.

1. **Matching 'presidential' multiple times in a document to be given more preference**

For that, we need **term frequency (TF)** vectors instead of bit vectors in our simple VSM. This change presents more information about the document.



Does this new representation of vectors solve our problem? Let's see the example again.



Vector 'q' still is a bit vector, in a way since all words occur only once. The change is in vector of d4 as now presidential word, which occurred twice is given importance in the term frequency representation of this vector.

given importance in the term frequency representation of this vector.

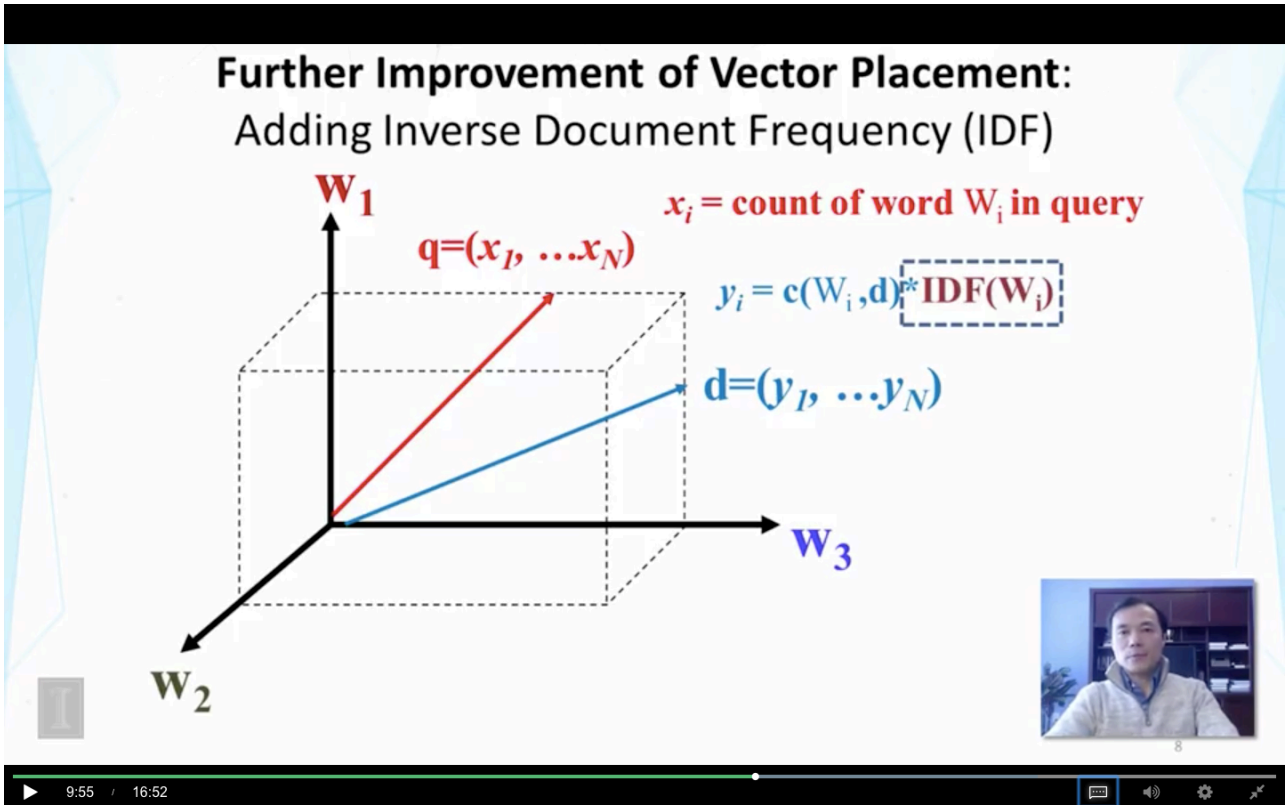
We can see, from the example, that d4 is now ranked above d2 and d3. However, d2 and d3 still have the same scores. We need to address the problem that d2 is irrelevant since it does not talk about 'presidential'.

1. **'presidential' vs 'about'**

A term like 'about' occurs very often in documents and across documents too, as in collections. We need to somehow reduce the weight of 'about' in the vector representation. Then, d2 will have a score less than 3 while d3 have a score of more than 3 and we will be able to achieve the ranking that we desire.

We modify the vector placement by including something called **Inverse Document Frequency (IDF)**. We know document frequency is the no. of documents in the collection where the word occurs. If DF is high meaning the word occurs in many documents, which probably means that the word is common word like 'about'. We want to penalize the term frequency of such a word. Hence, we invert the document frequency.

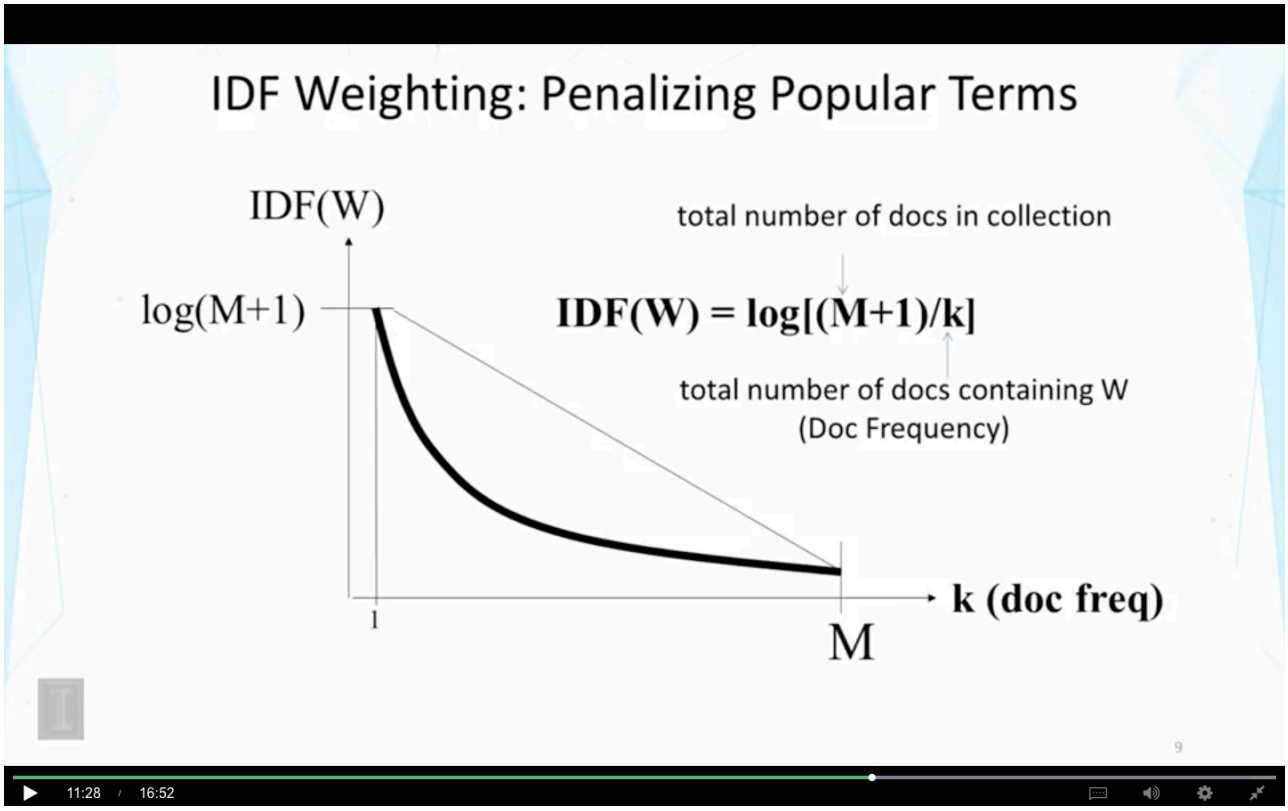
Why IDF? Because we want to reward a word that does not occur often in documents.



We achieve this by multiplying IDF with the term frequency.

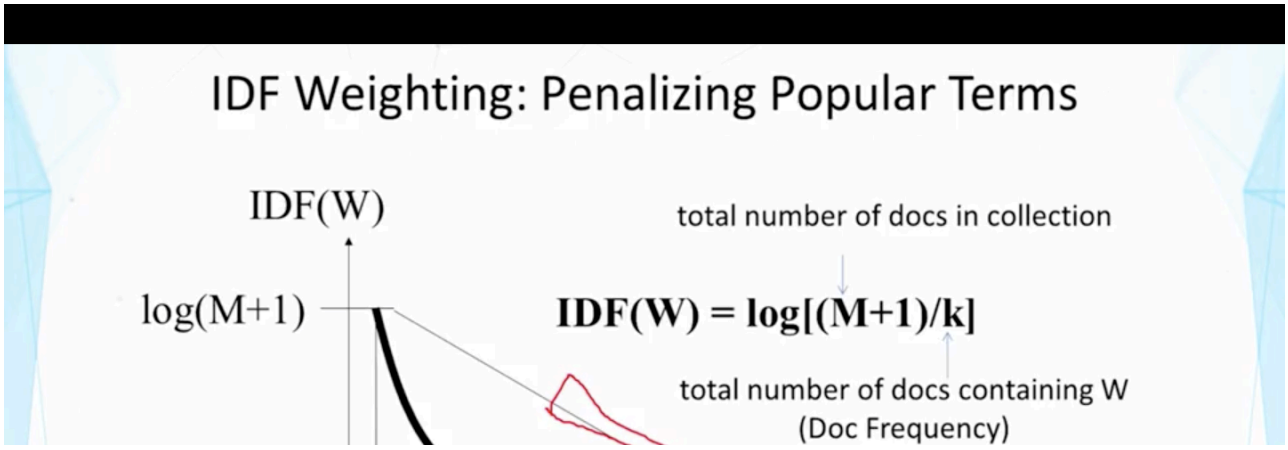
Observations:
Common words (‘about’) - occur in more documents - Low IDF
Rare words (‘presidential’) - occur in less documents - High IDF

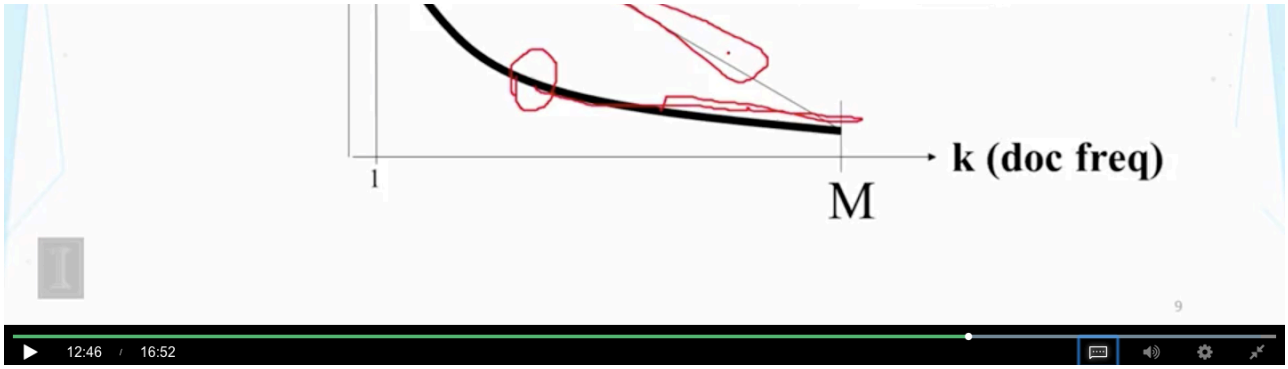
So, IDF of a word is calculated as:



We can see that IDF of a word when k is low is high, which essentially means that a word with low document frequency will have higher weight which is exactly what we want.

This function has been the standard for calculating IDF. Whether there is a better function than this is an open research function.
However, let’s just compare standard IDF v/s Linear IDF.

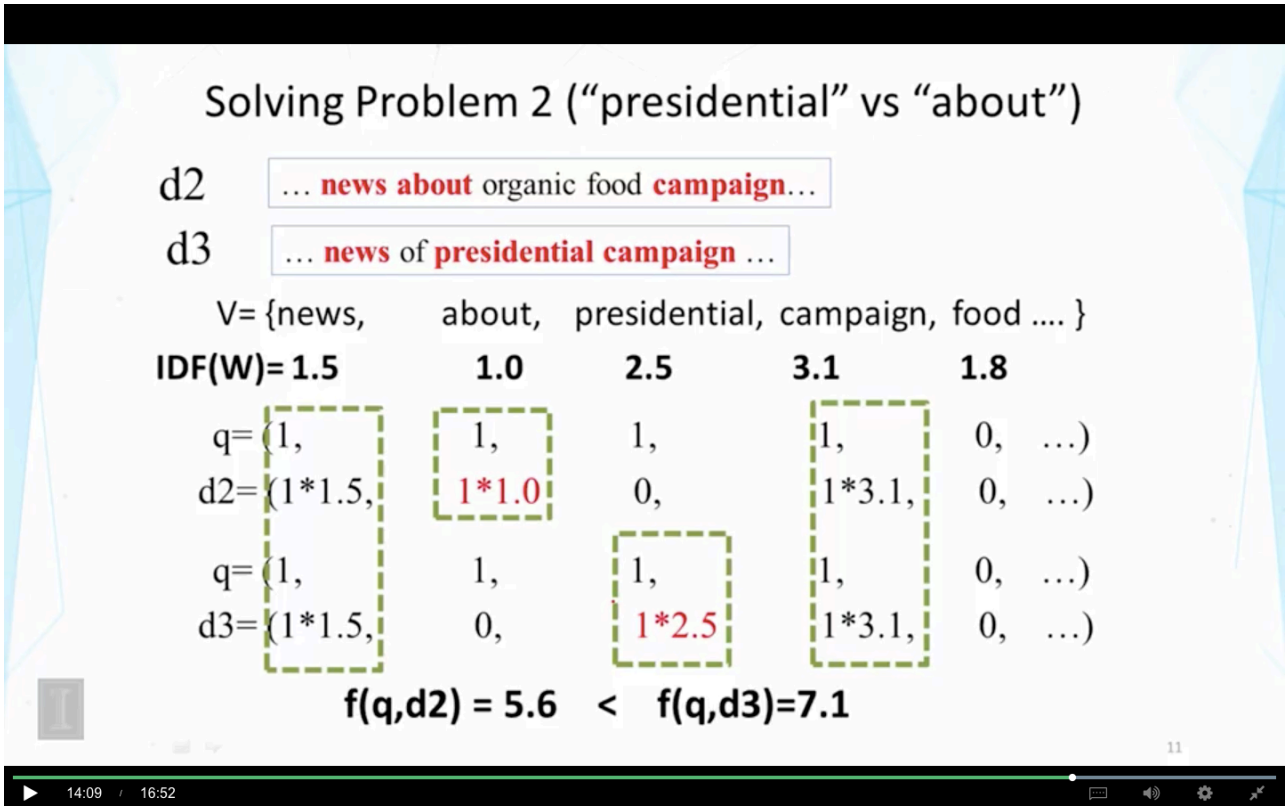




We can see that there is a turning point in the standard IDF. Basically, after that point, the IDF of all words with high k will be essentially very low as all these words will be common words and should not be given importance.

However, the linear IDF does not differentiate between the two. There is still a major difference between the points in the rectangle on linear IDF. The efficient of these 2 IDF functions needs to be empirically tested.

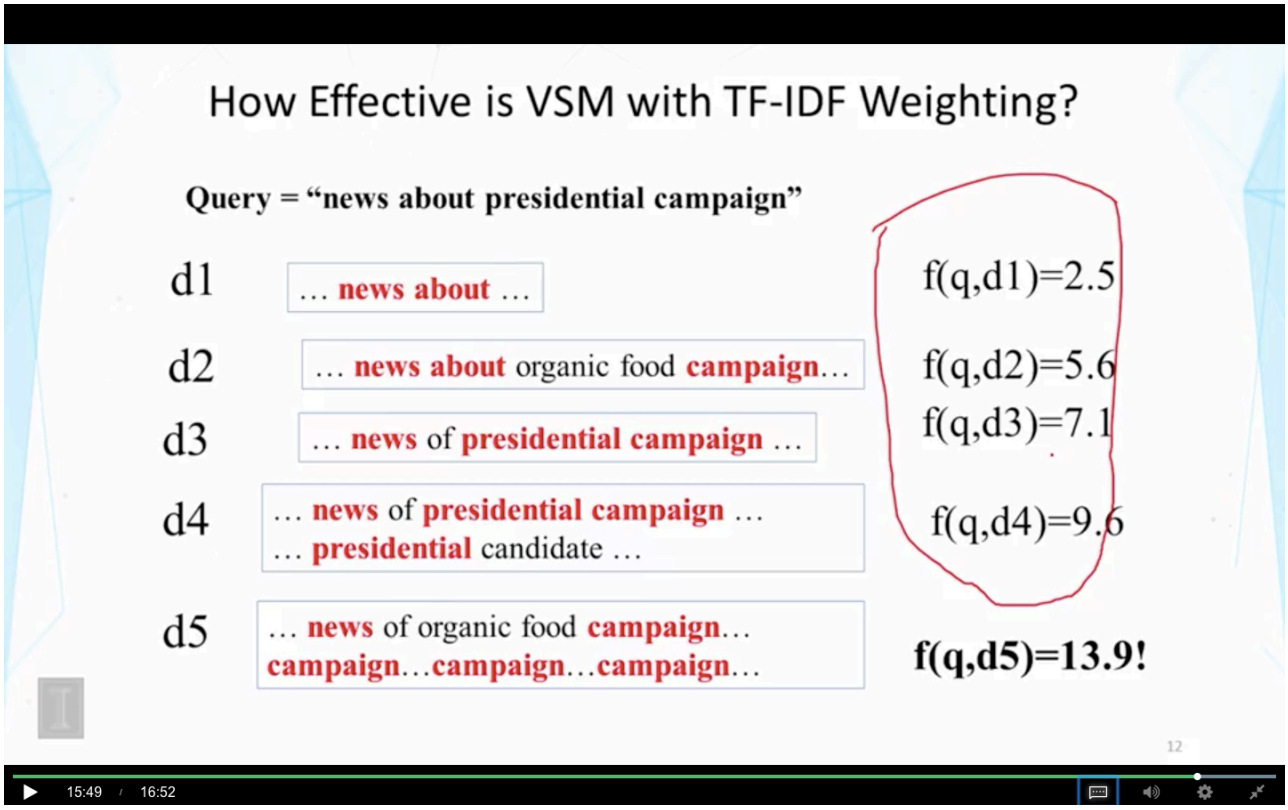
So, let' see if the TF-IDF approach is helpful in solving the 2nd problem or not.



Here, we see that IDF('about') < IDF('presidential'). Since, we multiplied these IDF values to the term frequency vector elements, it basically means that the importance of 'presidential' is more than 'about'. These means that the ranking function value of d3 is more than d2, hence d3 will be ranked

the ranking function value of d5 is more than d2, hence d5 will be ranked above d2 and this is the desired behaviour.

How effective is the VSM with TF-IDF weighting?

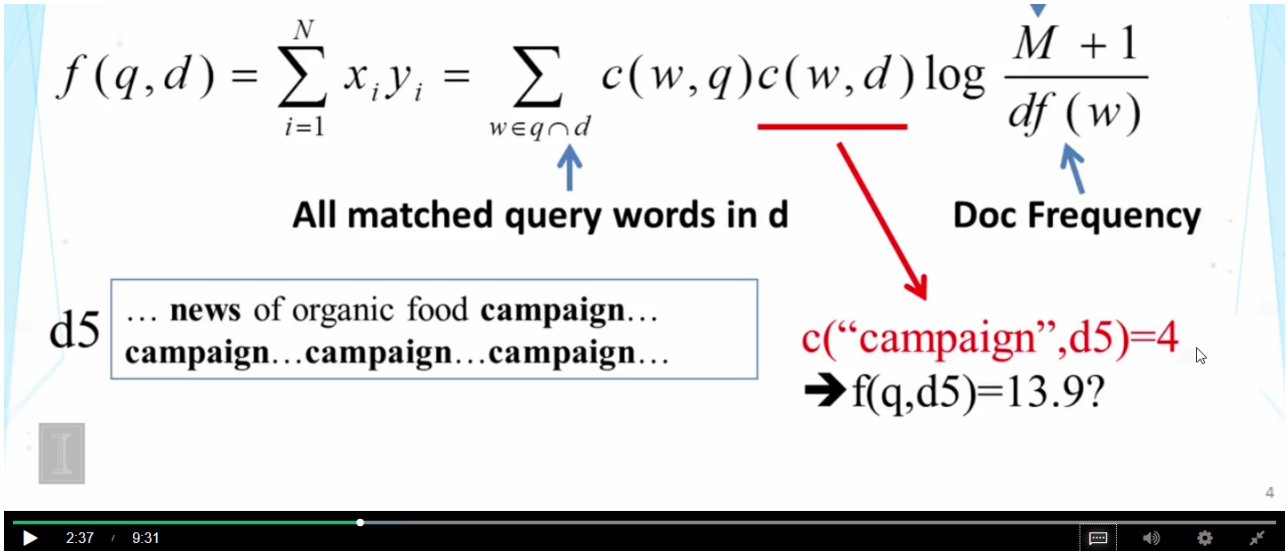


We see that the ranking function scores for the first 4 documents is reasonable however, the score of the 5th document is very large and this leads to improper ranking. Hence, we need to incorporate some more ideas to further improve this model.

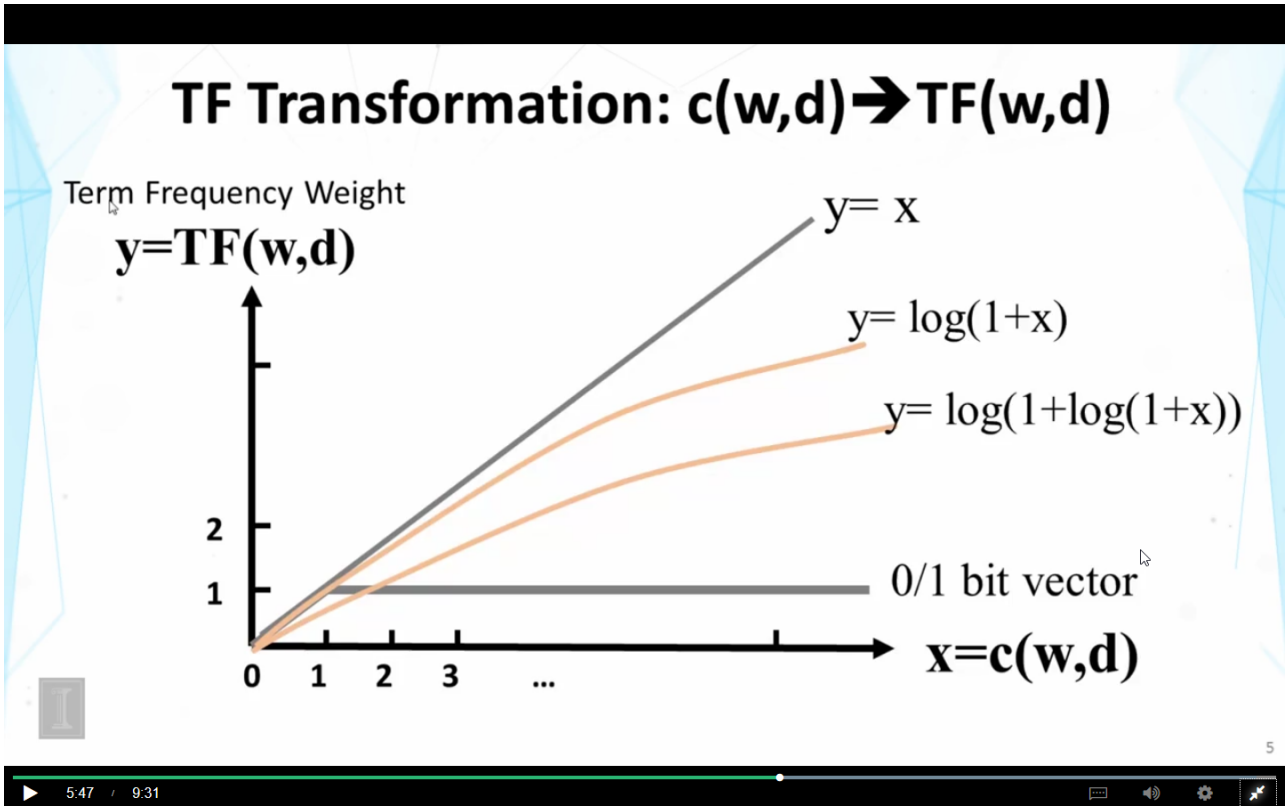
————————————————————————

Lecture 2.2 - (TF Transformation):

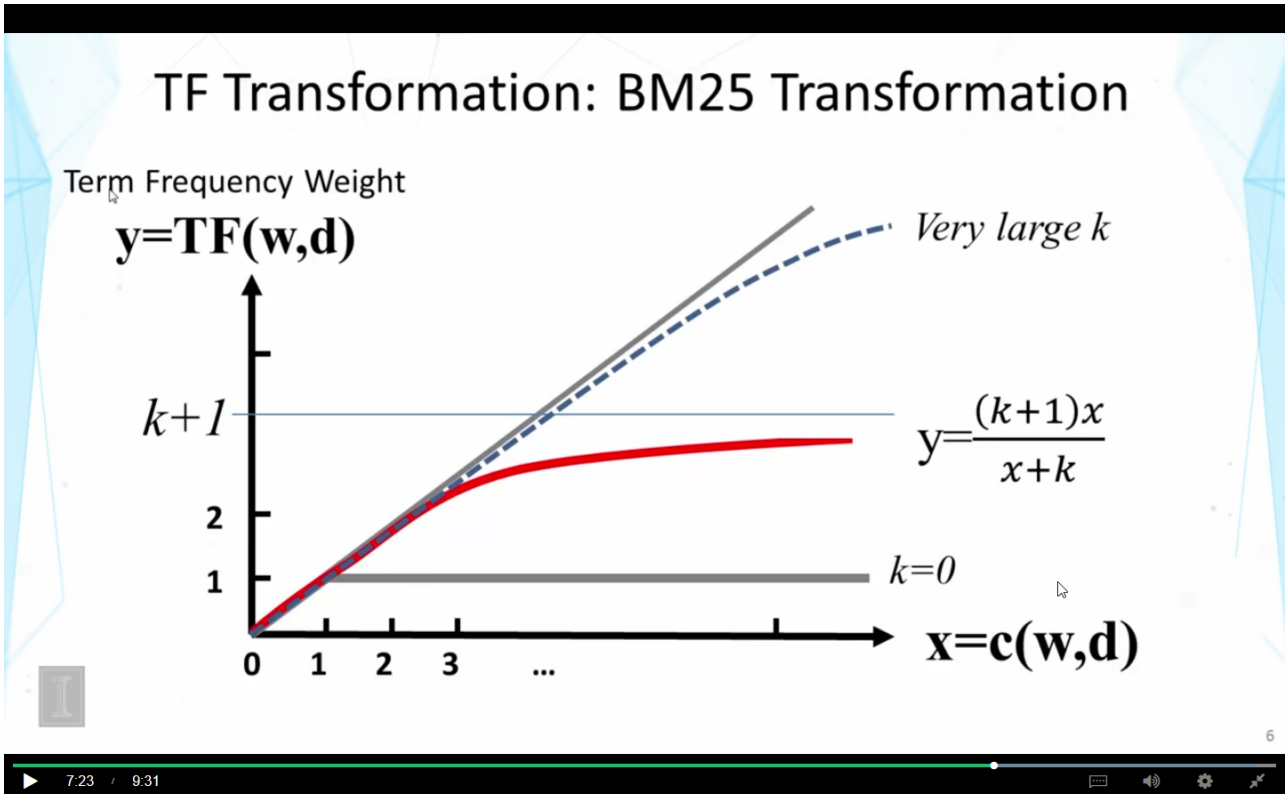
As we saw in the last lecture, the ranking function scores the document 'd5' high. So, why does this happen? For this, we need to look at TF-IDF ranking function.



Since, count of word in document is high, so is the ranking score. This happens because we have generously rewarded points for multiple occurrences of words. When count of word increases from 0, it is a big step because it shows that document is talking about the word. However, if count increases from 50 to 51, it does not prove anything. This can also lead to the spamming scenario. So, we change the count vector.



We can use 'log' functions which control the term frequency vector when count is high.

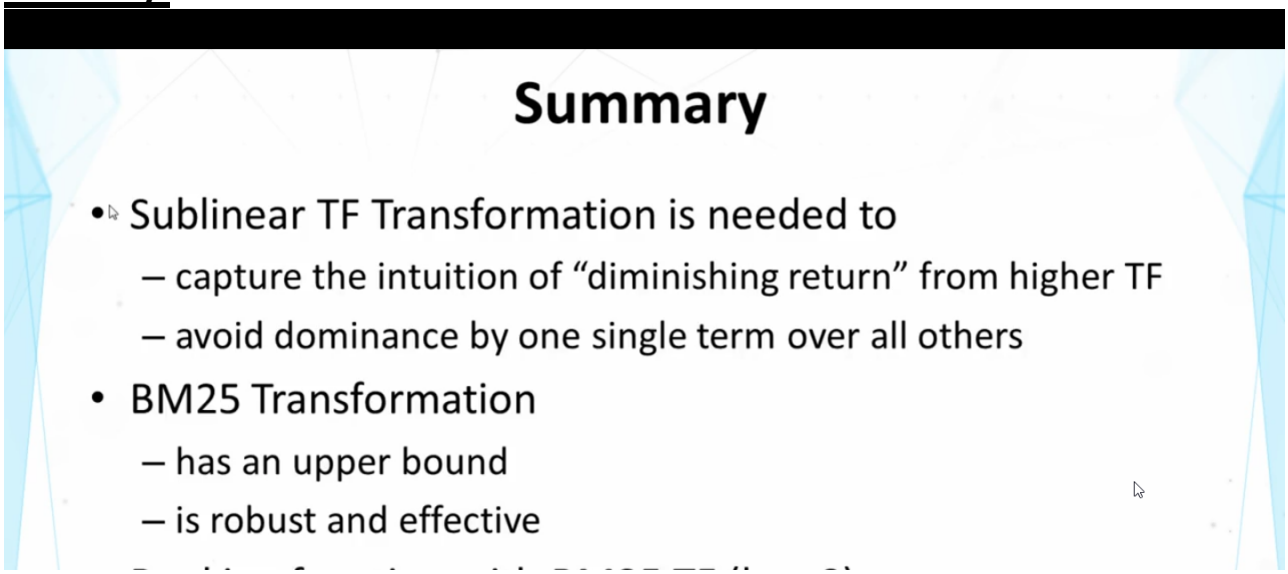


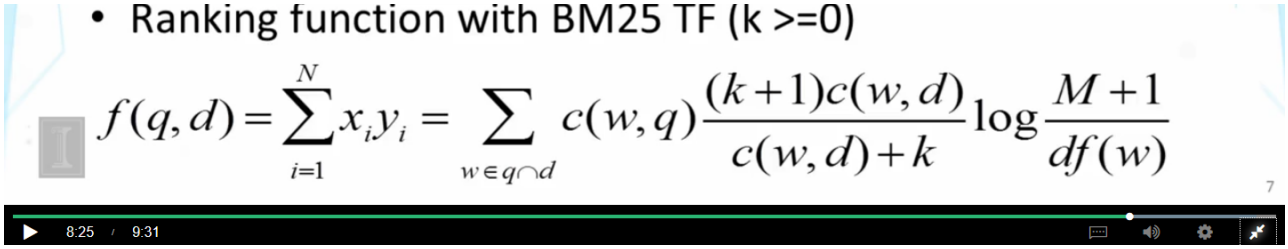
However, we see that the BM25 transformation function has worked well till now. It is robust and effective. Now, this function has an extra parameter 'k'. This 'k' value provides the upper bound for the function and can also generate the bit term or linear term vectors.

- Ranking function with BM25 TF (k >=0)

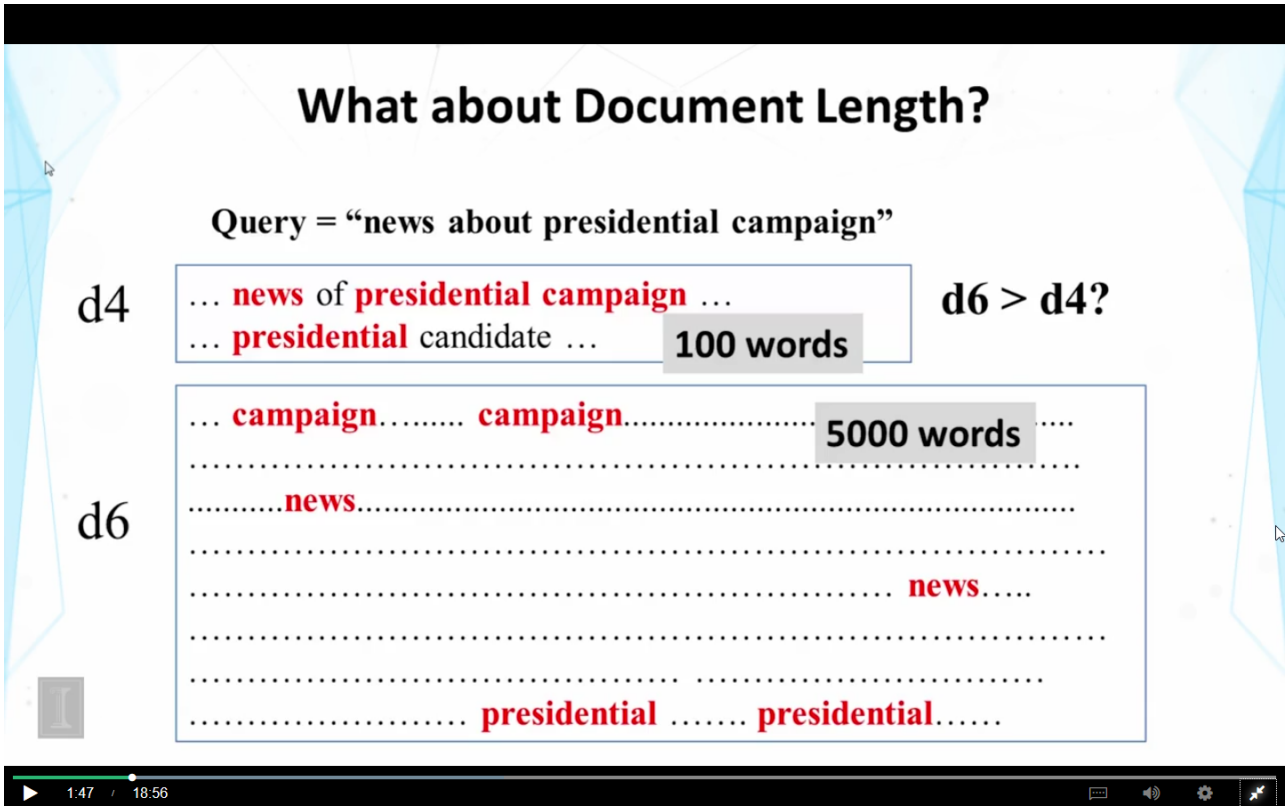
$$f(q,d)=\sum_{i=1}^{N} x_i y_i = \sum_{w\in q\cap d} c(w,q)\frac{(k+1)c(w,d)}{c(w,d)+k}\log\frac{M+1}{df(w)}$$

Summary:

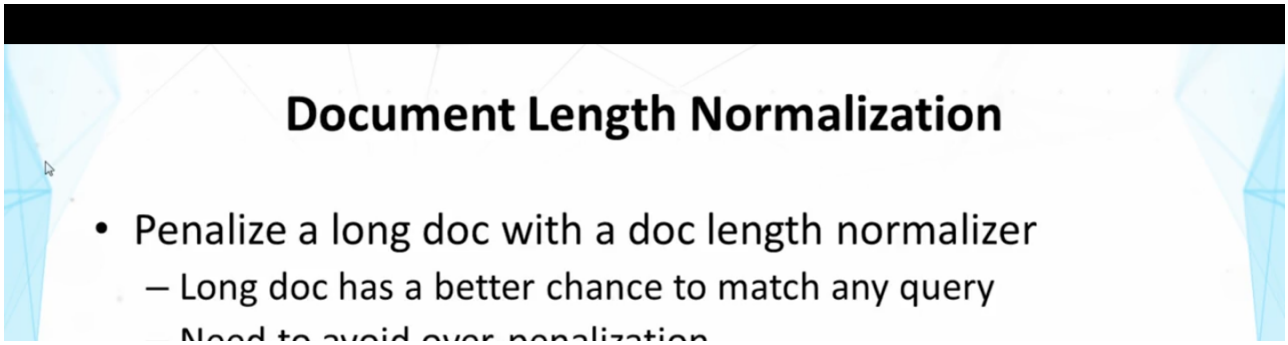


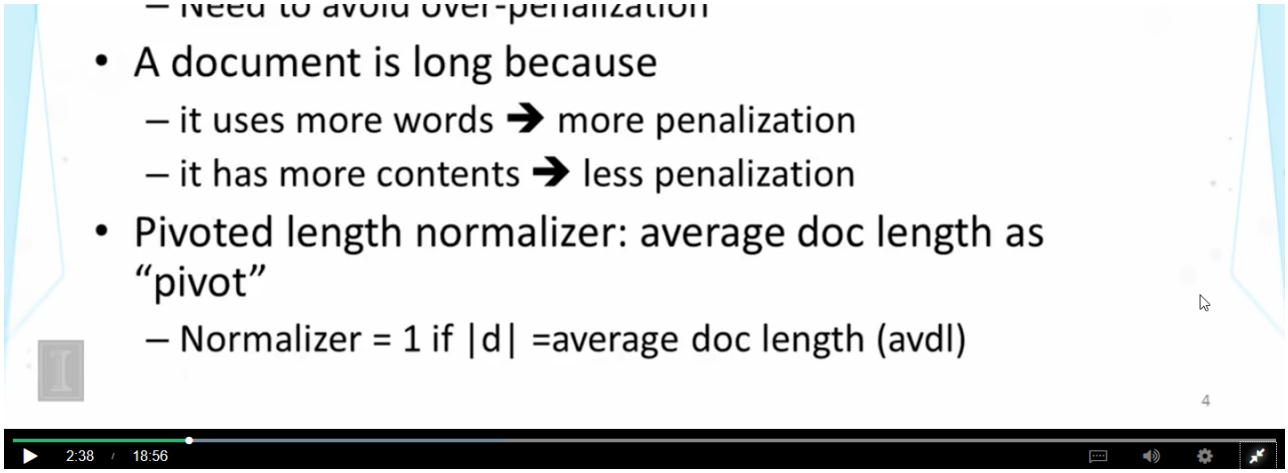


Lecture 2.3 - (Doc Length Normalization):



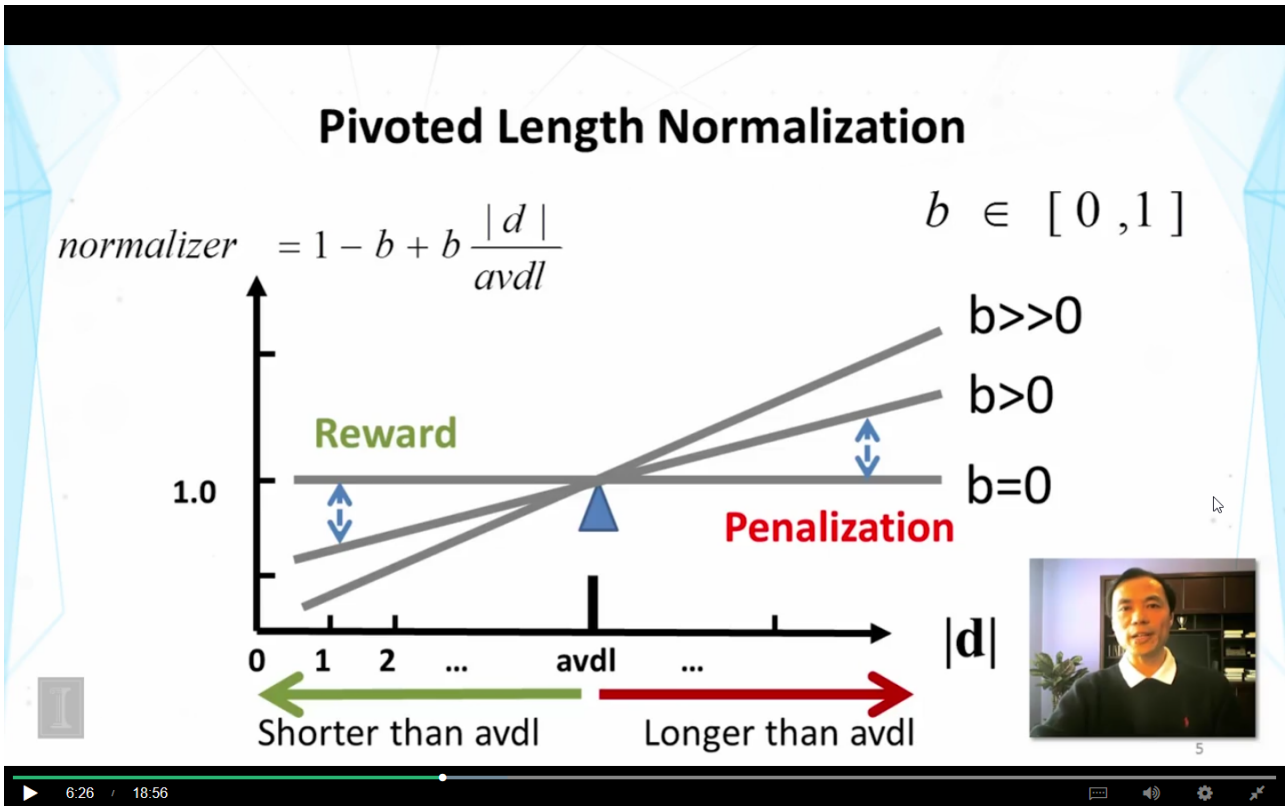
The score of word also depends on length of documents.
Long texts contain:
More amount of words
Scattered words which means the content may not be much relevant. So we need to find a way to penalize the documents based on their lengths.





Although, documents can be long for different reasons. So, we need to take care whether document has more relevant or more non-relevant words.

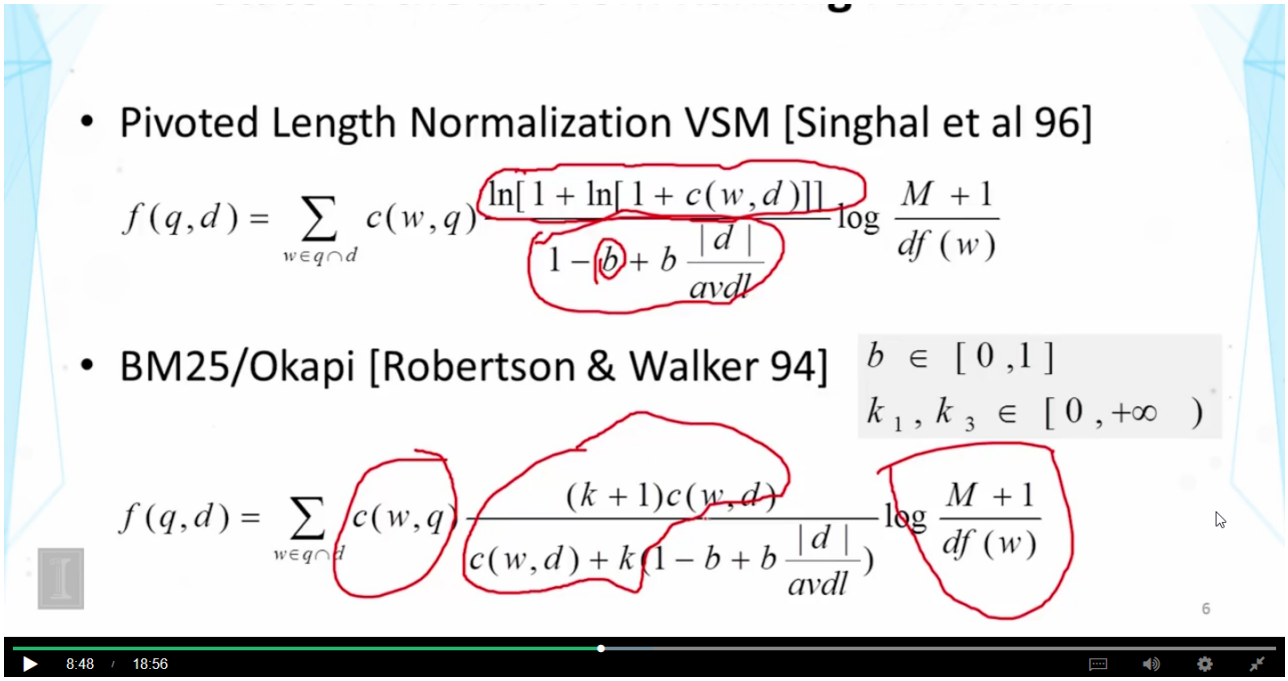
A normalizer which is well-tested is pivoted length normalizer. It is defined as:



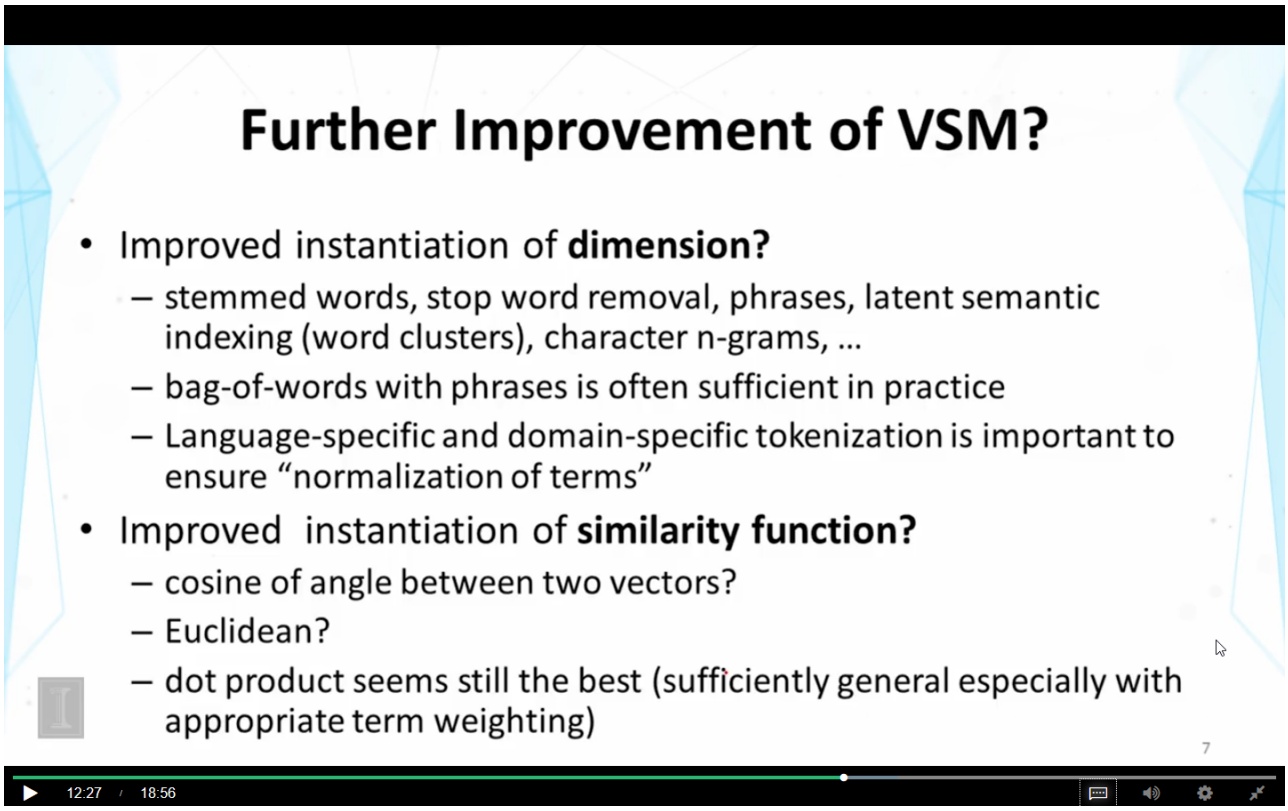
'b' - parameter which measures the degree of penalization for longer documents.
- also represents slope of line in normalizer equation

Document length divided by average doc. Length to compare and also to make it unit-independent.

State-of-the-Art ranking functions:



Pivoted length - 'double log' TF transformation
BM25/Okapi - 'BM25' TF transformation



Dimension:
Currently, each word as an dimension and generate vectors for query and documents.
However, we can include different ideas like:
Stemmed words - reduce words to their '**roots**', so they can be matched, basically a

way for approximate matching
Stop words - common words: 'the', 'a', 'of', 'for'
Phrases
N-gram representation
Latent semantic indexing techniques

Bag-of-words representation with phrases is most efficient.

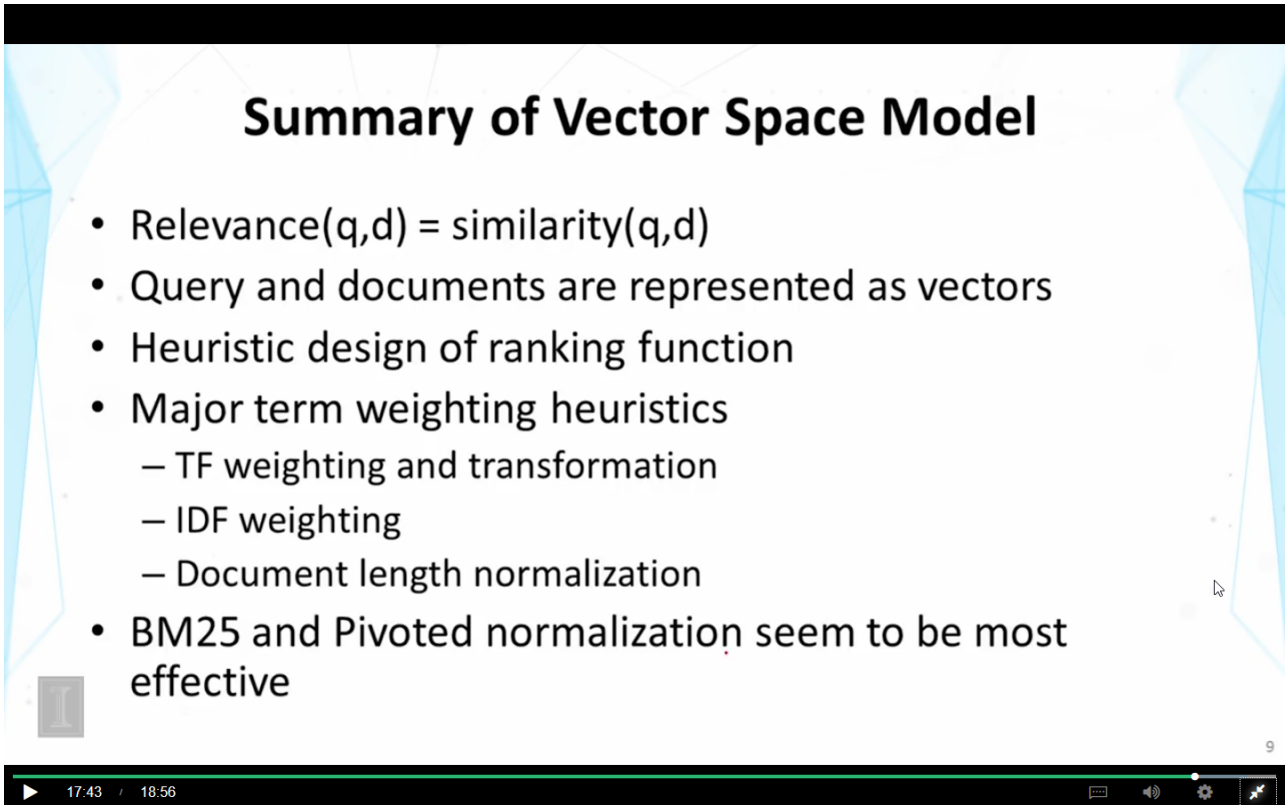
Similarity function:

We have used **dot product** measure, however, we can use:

1. Cosine of angle b/w vectors
2. Euclidean distance

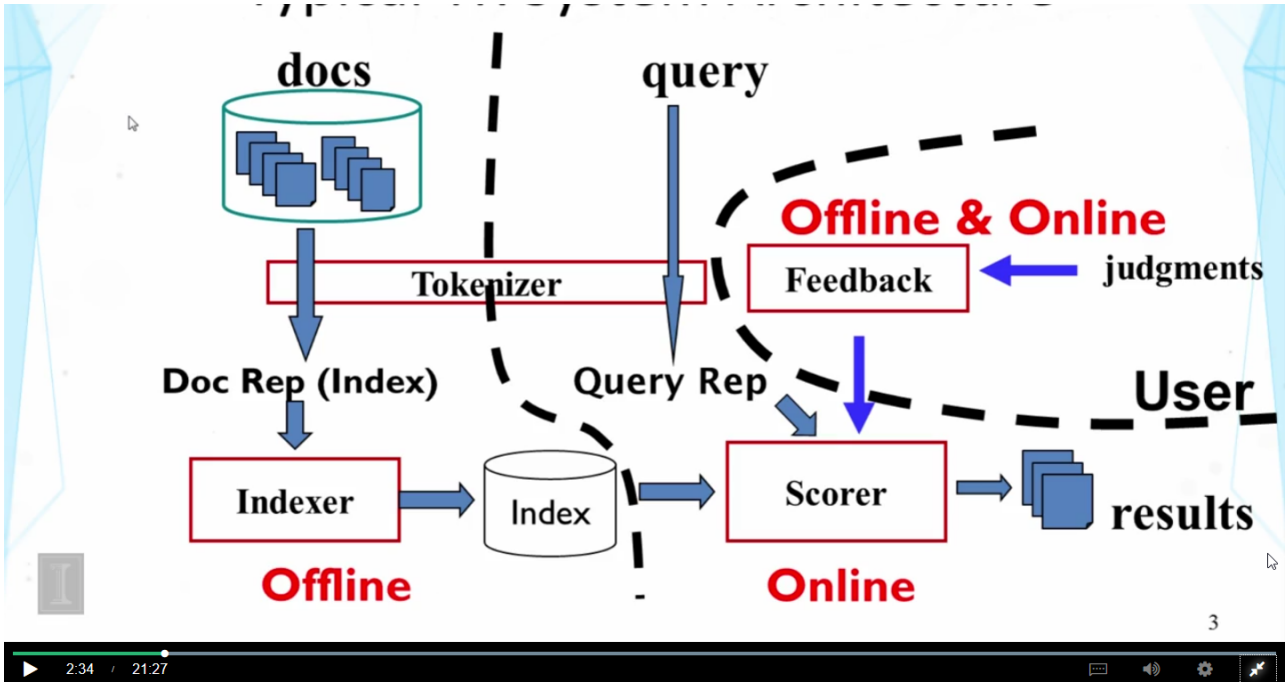
Dot product is sufficiently general with apt term weighting techniques.

Summary:

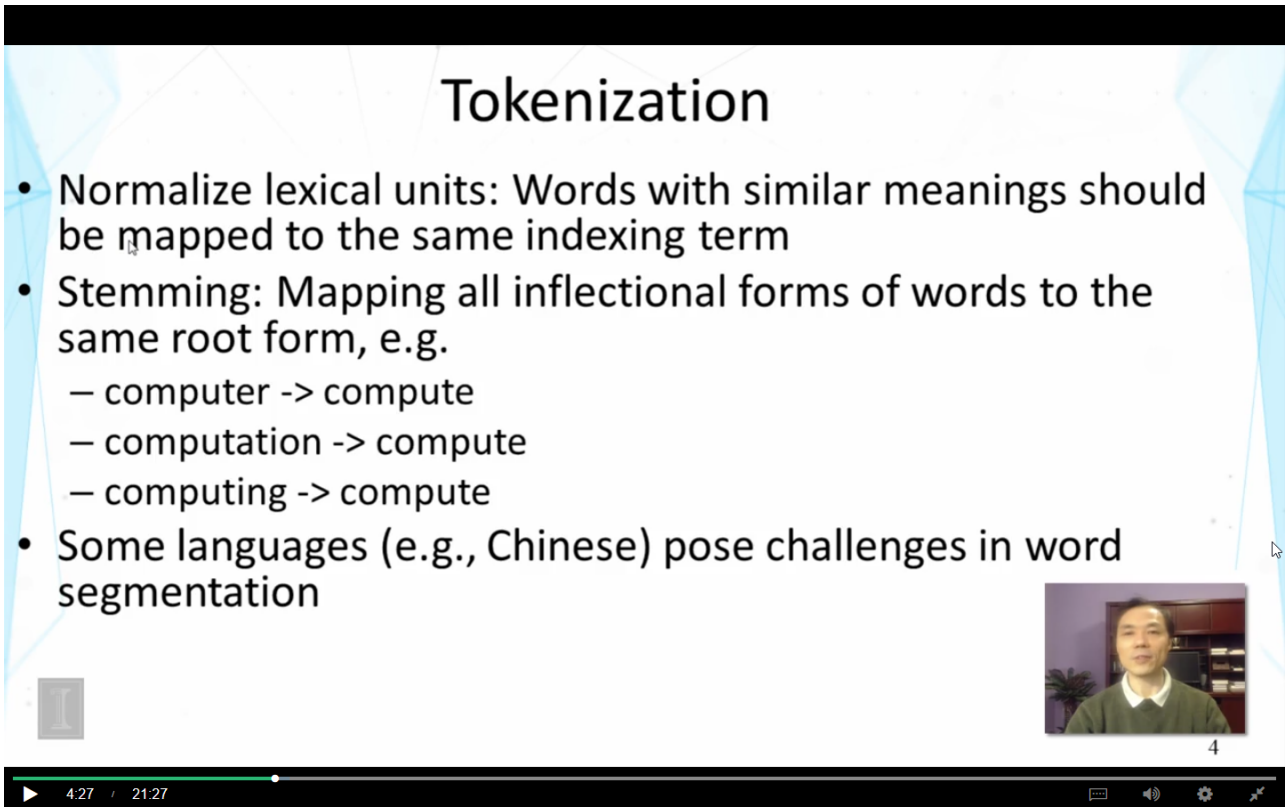


———————————————————

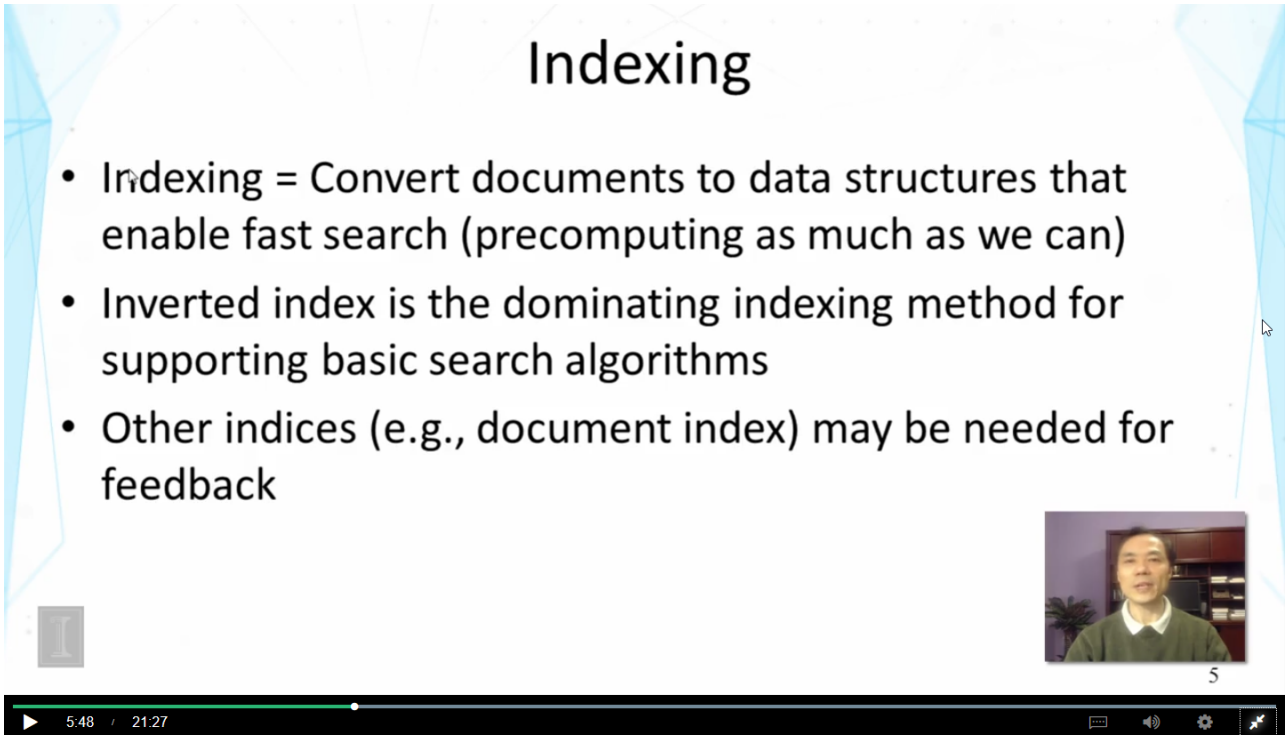
Lecture 2.4 - (Implementation of TR systems):



The documents are passed through the tokenizer to decompose the documents into words. It also stems the words so that common root words match.

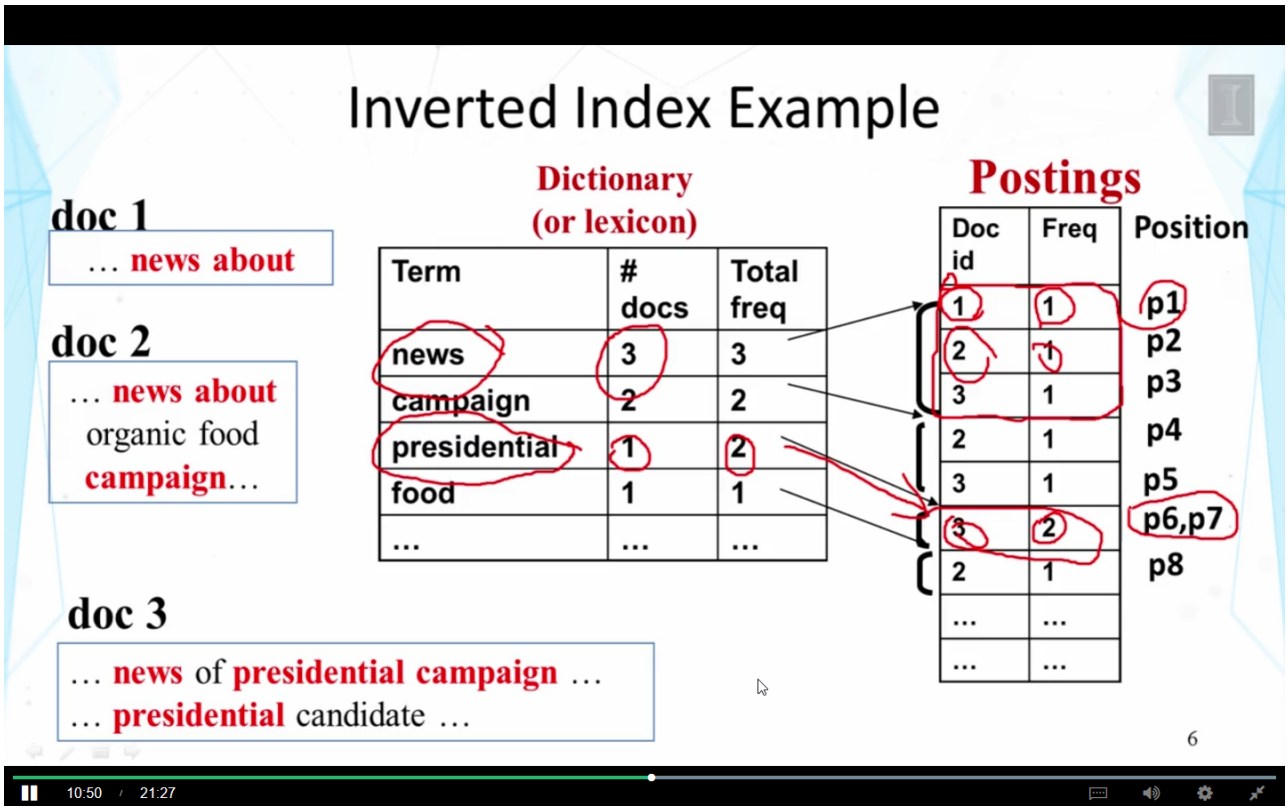


This is then passed to indexer which indexes the words, their frequencies and document nos. so as to enable fast access.



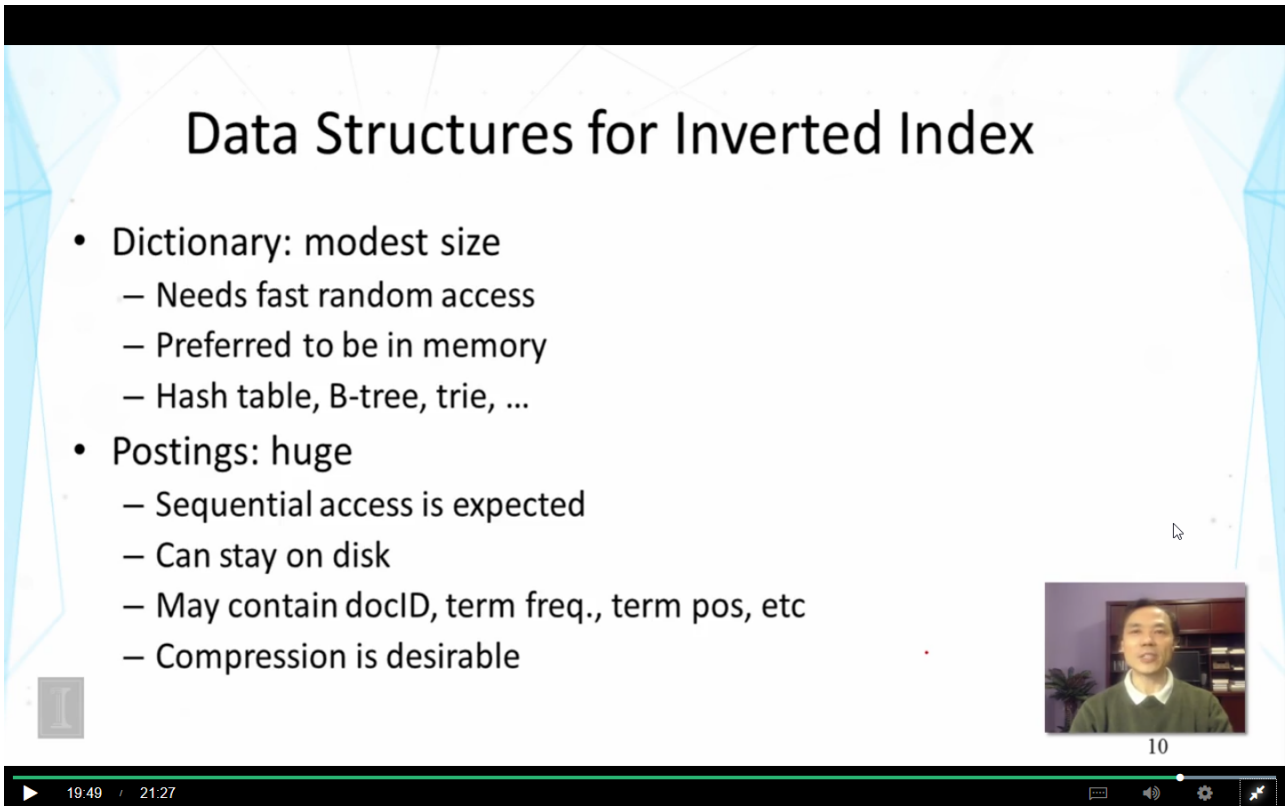
The query is also tokenized and then score is calculated. The user provides implicit or explicit feedback on the results.

Indexing - inverted index - -better and fast for search



The dictionary contains the words found in the documents which have high

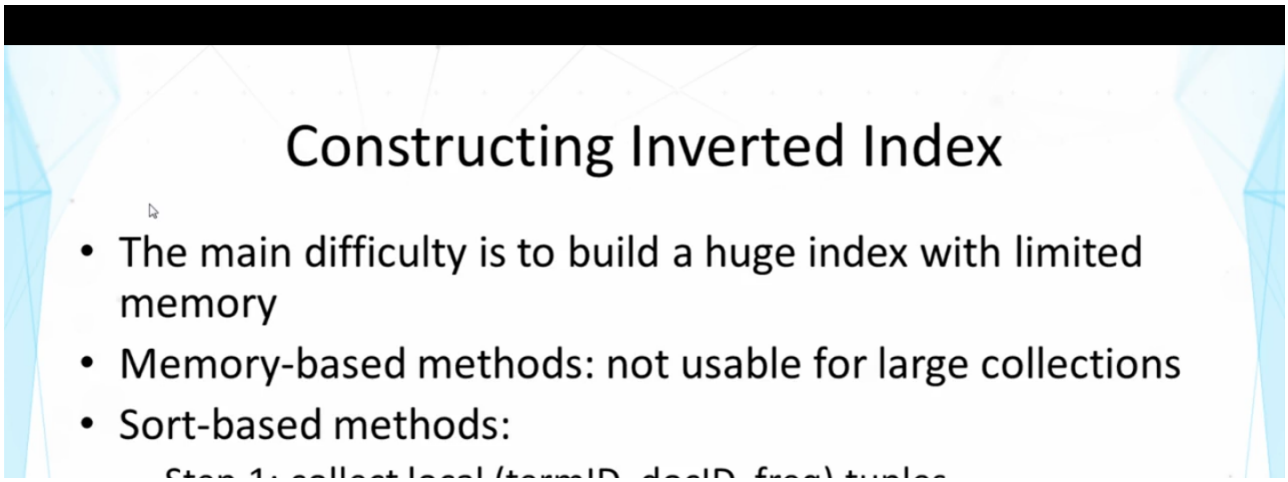
IDF (low IDF, remove stop words). The no of documents and frequencies are stored. Each row in the dictionary contains a pointer to another table which has the document Ids, their positions and frequencies of these words. The documents containing a word are stored sequentially so as to fetch documents quickly for a single term.

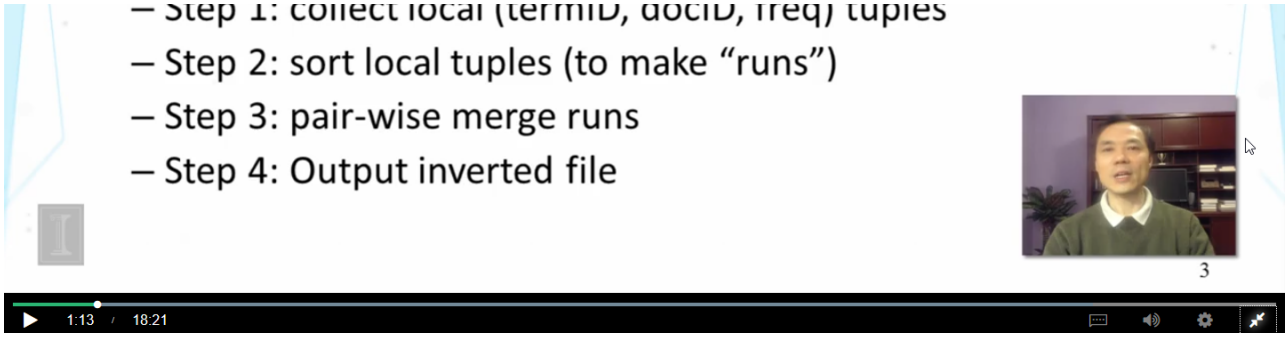


Dictionary - direct access - fast - hash table, B-Tree, Trie - In Memory
Postings - sequential access - compression desirable bcoz huge size - Disk storage

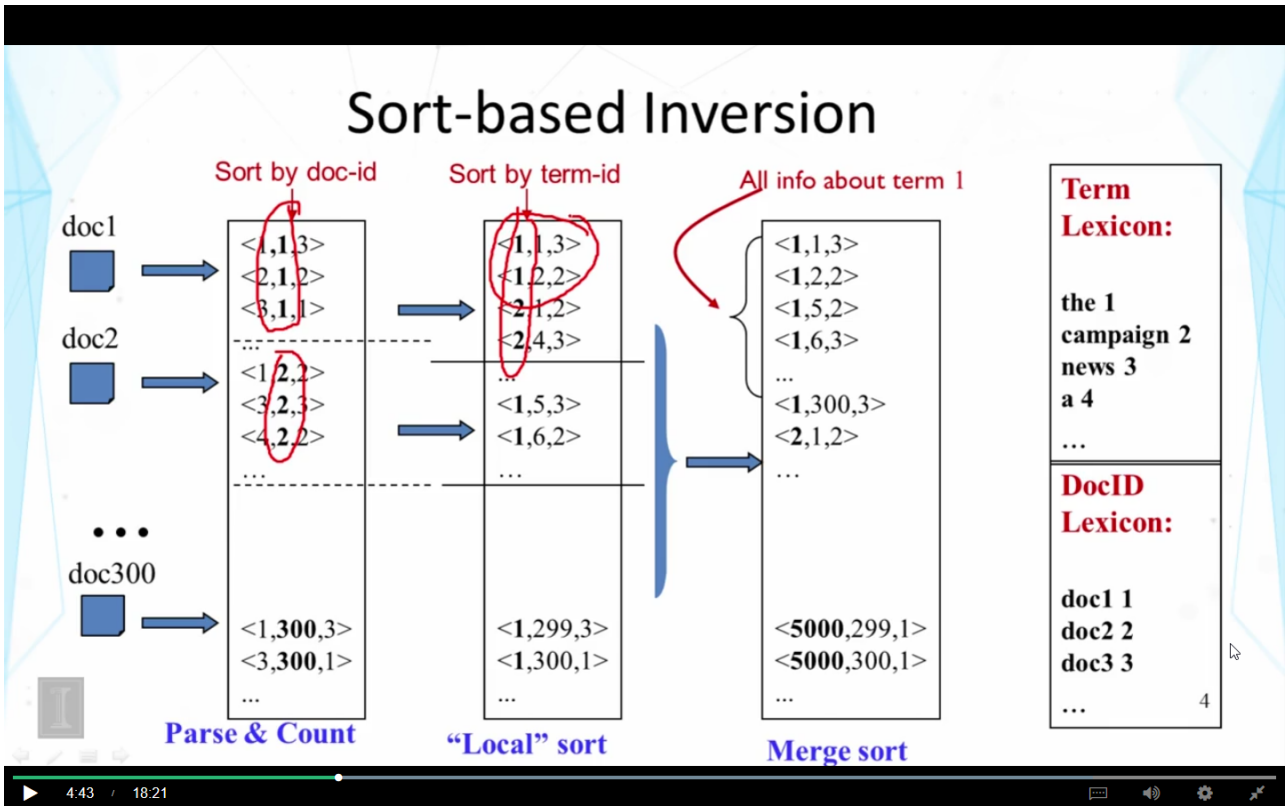
————————————————————

Lecture 2.5 - (Inverted Index Construction):

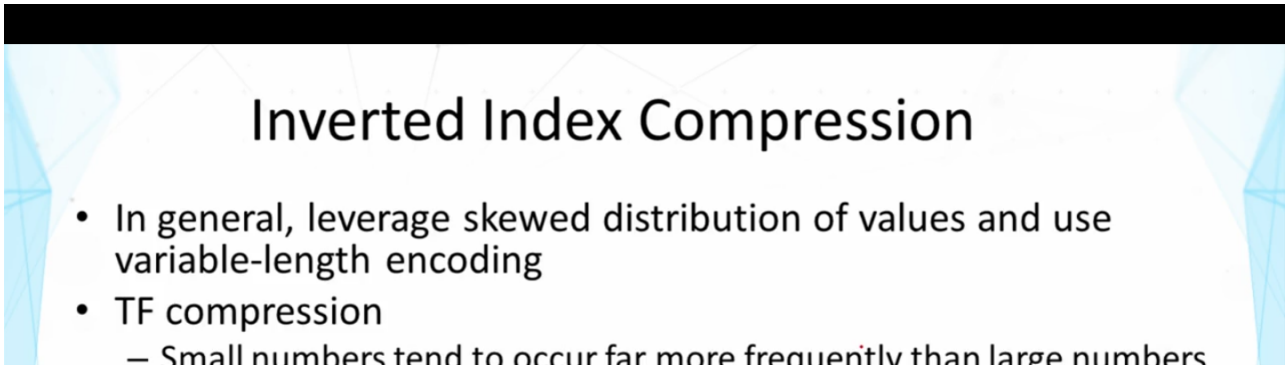


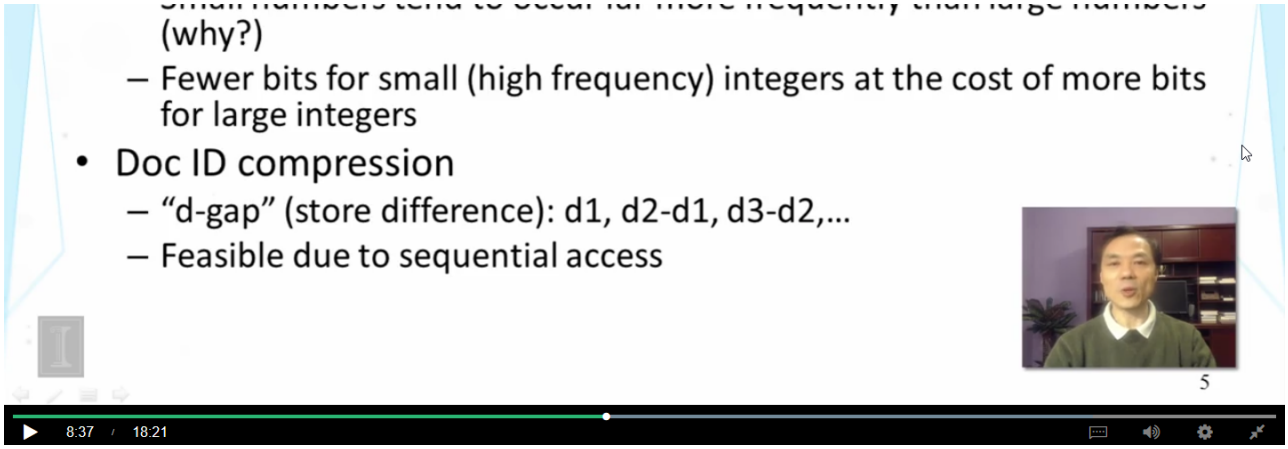


Data is very large in text retrieval problems, no. of documents is high, so we cannot simply construct an inverted index from all of the documents and store it in memory. So, we break the storing of index into different rounds.



We first map the **terms** and **documents** to integers bcoz handling integers is easy and we need integer representation for index compression. We use many temporary files to store temporary outputs after parsing and sorting.

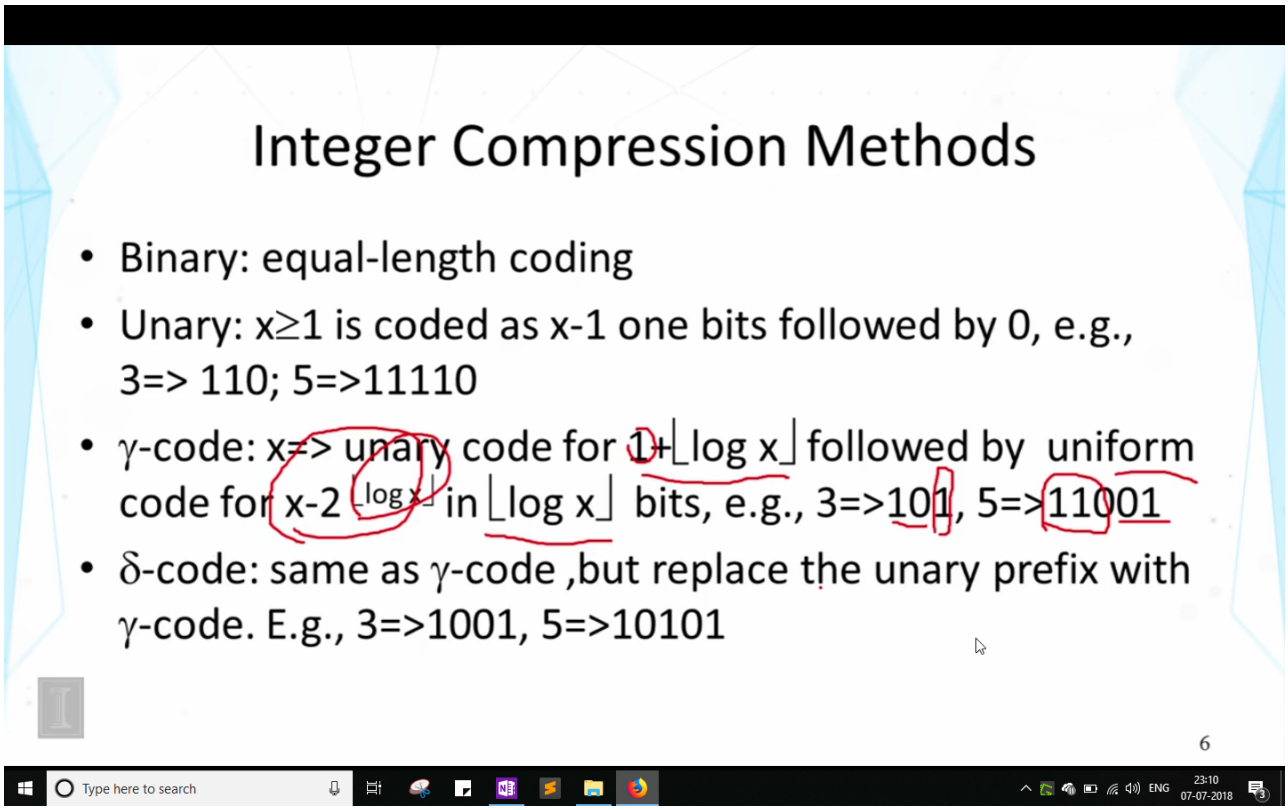




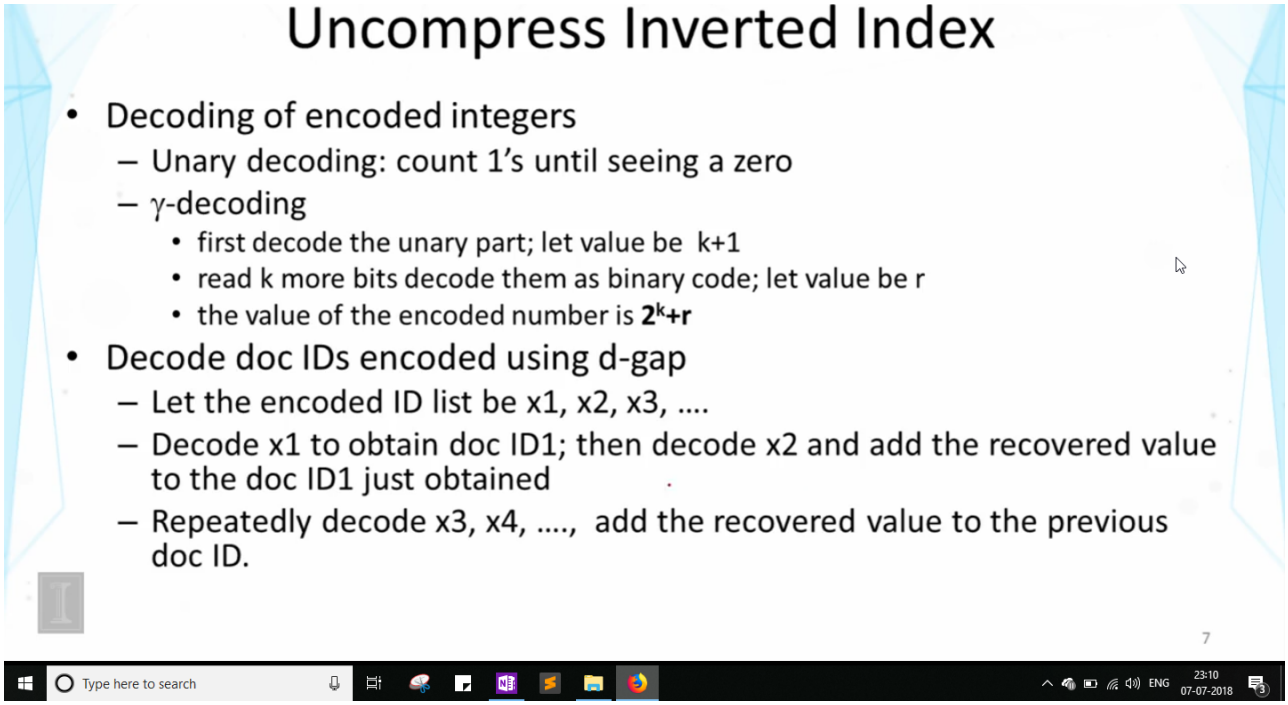
Since, inverted index table can be huge, we need to compress the table and that's why we have integer representations of doc ids and term ids.

Smaller count values for term frequencies are more frequent, of course. So, we need less bits to store smaller numbers and more bits to store large numbers.

Document ids can be compressed as a 'd-gap' mechanism. What we do is keep the 1st document id and then store the difference of current doc id and previous doc id. This is possible bcoz doc ids are sequentially stored.



Variable length encoding for integers



Decoding can be done in the opposite way it was encoded.

Lecture 2.6 - (Fast Search):