

# Week 2

## Lecture 2.1 - (VSM - Improved instantiation):

The slide has a light blue geometric background. At the top center, the title 'Two Problems of the Simplest VSM' is displayed. Below the title, the query 'Query = “news about presidential campaign”' is shown. Three documents are listed with their term frequencies:

Document	Text Content	f(q,d)
d2	... news about organic food <b>campaign</b> ...	f(q,d2)=3
d3	... news of <b>presidential campaign</b> ...	f(q,d3)=3
d4	... news of <b>presidential campaign</b> ... ... <b>presidential</b> candidate ...	f(q,d4)=3

Below the table, two points are listed in a box:

1. Matching “**presidential**” more times deserves more credit
2. Matching “**presidential**” is more important than matching “**about**”

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other presentation controls.

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?

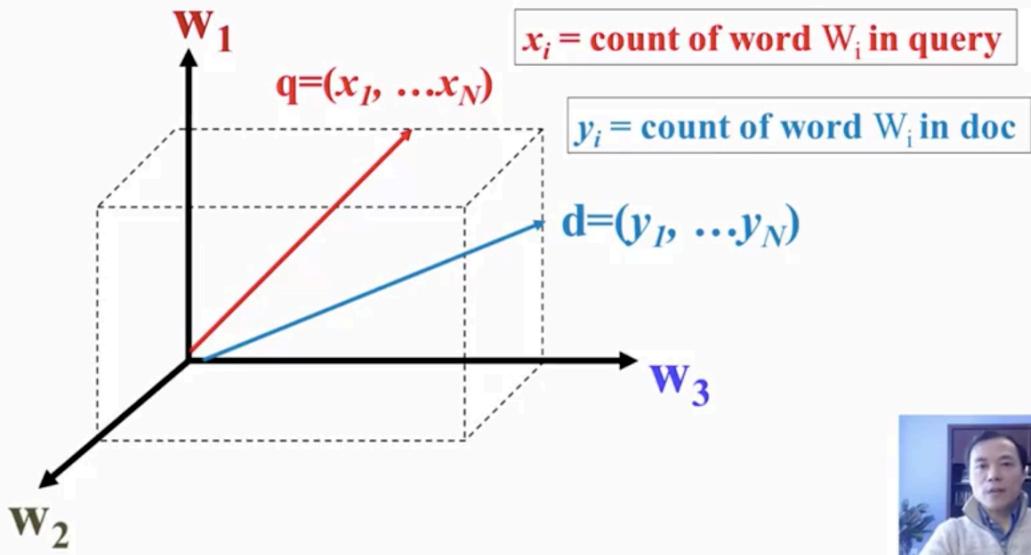
These problems exist really from the assumptions that we have made in placing vectors in the vector space. So, we need to have different ways to instantiate vectors in the space.

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.

## Improved Vector Placement: Term Frequency Vector



3:59 / 16:52

[play] [stop] [volume] [settings]

4

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

### Ranking using Term Frequency (TF) Weighting

$$d2 \quad \dots \text{news about organic food campaign...} \quad f(q,d2)=3$$

$$\begin{array}{ccccc} q= & (1, & 1, & 1, & 0, \\ d2= & (1, & 1, & 1, & 0, \end{array} \dots) \quad \dots)$$

$$d3 \quad \dots \text{news of presidential campaign...} \quad f(q,d3)=3$$

$$\begin{array}{ccccc} q= & (1, & 1, & 1, & 1, \\ d3= & (1, & 0, & 1, & 1, \end{array} \dots) \quad \dots)$$

$$d4 \quad \dots \text{news of presidential campaign...} \quad f(q,d4)=4!$$

$$\begin{array}{ccccc} q= & (1, & 1, & 1, & 1, \\ d4= & (1, & 0, & 2, & 1, \end{array} \dots) \quad \dots)$$



6

6:27 / 16:52

[play] [stop] [volume] [settings]

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.

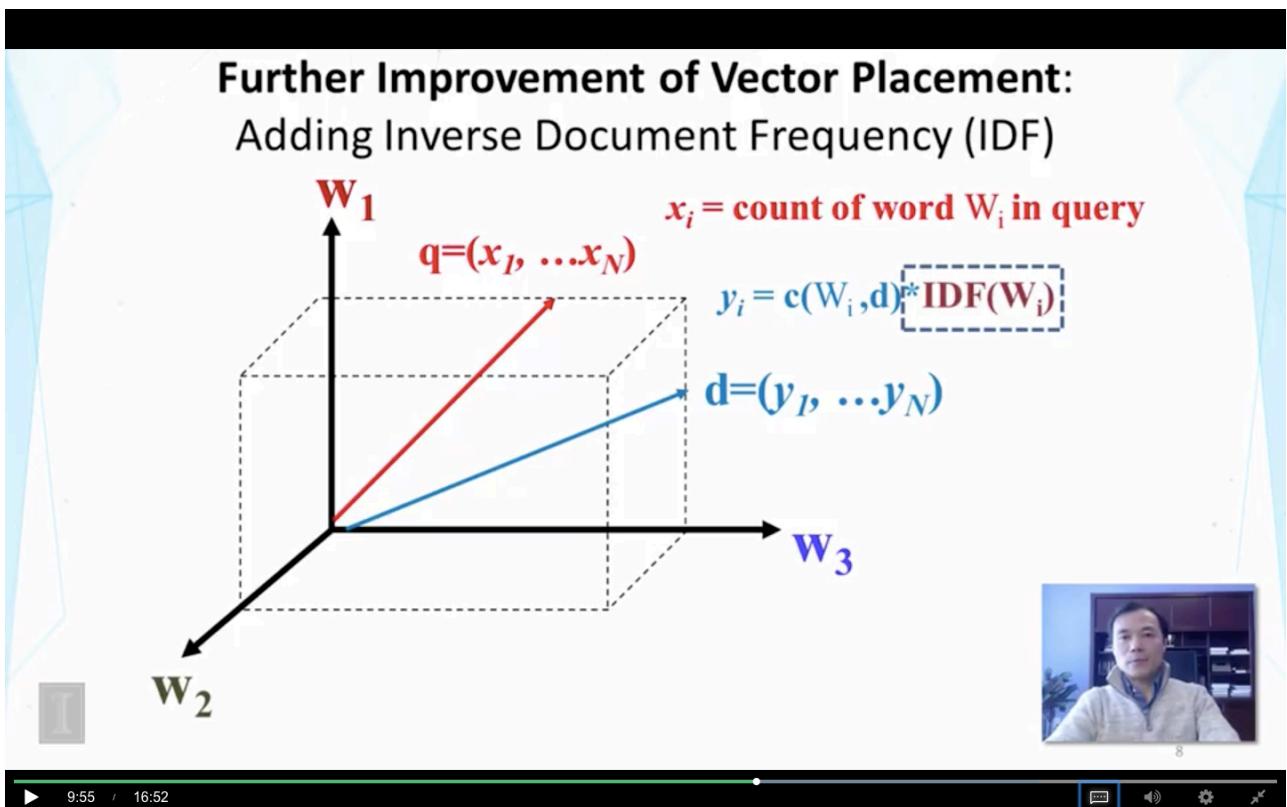
We can see, from the example, that  $d_4$  is now ranked above  $d_2$  and  $d_3$ . However,  $d_2$  and  $d_3$  still have the same scores. We need to address the problem that  $d_2$  is irrelevant since it does not talk about ‘presidential’.

### 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,  $d_2$  will have a score less than 3 while  $d_3$  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.



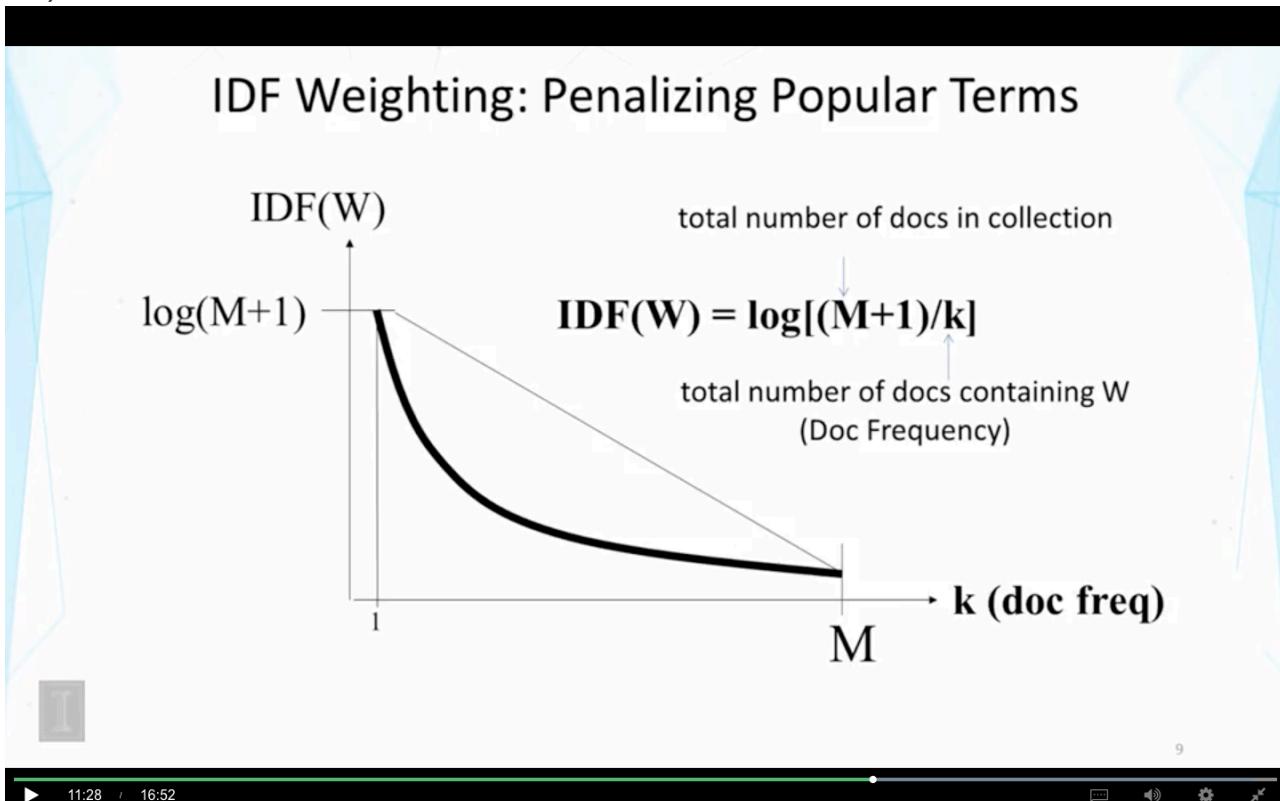
We achieve this by multiplying IDF with the term frequency.

Observations:

Common words ('about') - occur in more documents - Low IDF

Rare words ('presidential') - occur in less documents - High IDF

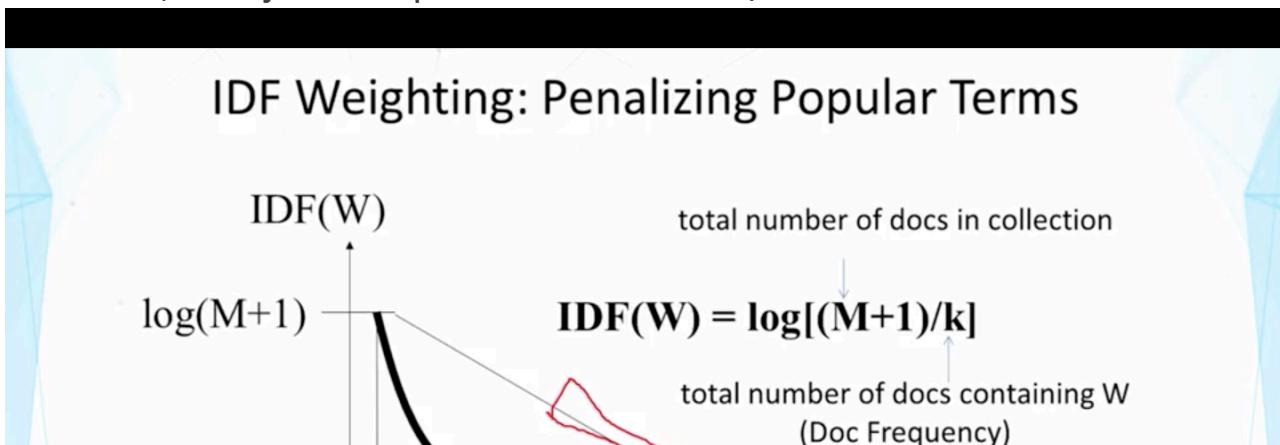
So, IDF of a word is calculated as:



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 question.

However, let's just compare standard IDF v/s Linear IDF.

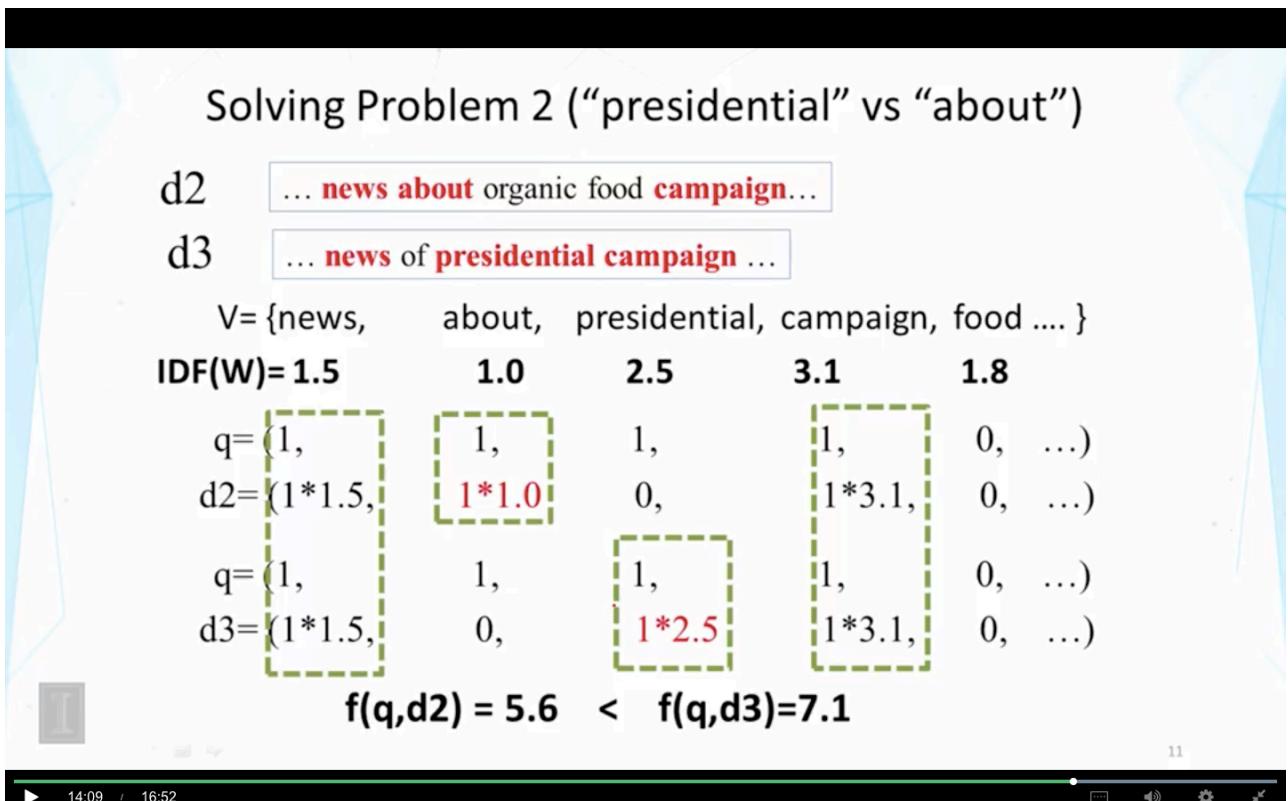




We can see that there is a turning point in the standard IDF. Basically, after that point, the IDF of all words with high k will be essentially very low as all these words will be common words and should not be given importance.

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's 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?

## How Effective is VSM with TF-IDF Weighting?

Query = “news about presidential campaign”

Document	Text Content
d1	... news about ...
d2	... news about organic food campaign...
d3	... news of presidential campaign ...
d4	... news of presidential campaign ... ... presidential candidate ...
d5	... news of organic food campaign... campaign...campaign...campaign...

$f(q,d1)=2.5$   
 $f(q,d2)=5.6$   
 $f(q,d3)=7.1$   
 $f(q,d4)=9.6$   
 $f(q,d5)=13.9!$

12

15:49 / 16:52

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.

## Ranking Function with TF-IDF Weighting

Total # of docs in collection  
↓

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

All matched query words in d

Doc Frequency

d5    ... news of organic food **campaign**...  
**campaign**...campaign...campaign...

$c(\text{"campaign"}, d5) = 4$   
 $\rightarrow f(q, d5) = 13.9?$

4

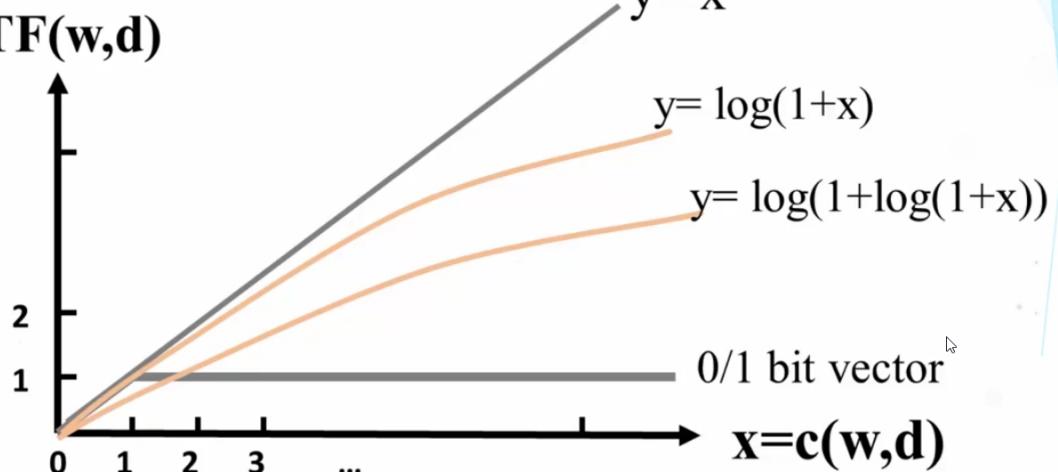


Since, count of word in document is high, so is the ranking score. This happens because we have generously rewarded points for multiple occurrences of words. When count of word increases from 0, it is a big step because it shows that document is talking about the word. However, if count increases from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

## TF Transformation: $c(w, d) \rightarrow TF(w, d)$

Term Frequency Weight

$$y = TF(w, d)$$



5

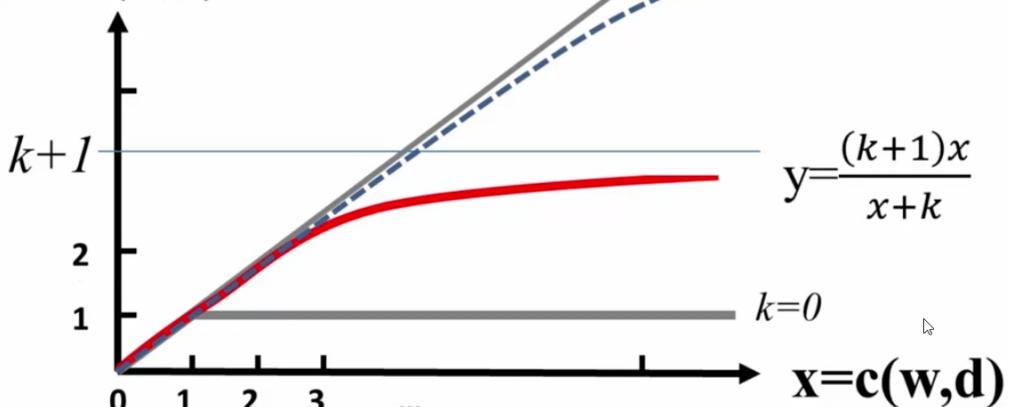


We can use 'log' functions which control the term frequency vector when count is high.

## TF Transformation: BM25 Transformation

Term Frequency Weight

$$y = \text{TF}(w, d)$$



6

▶ 7:23 9:31



...



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 ( $k \geq 0$ )

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

### Summary:

## Summary

- Sublinear TF Transformation is needed to
  - capture the intuition of “diminishing return” from higher TF
  - avoid dominance by one single term over all others
- BM25 Transformation
  - has an upper bound
  - is robust and effective

- Ranking function with BM25 TF ( $k \geq 0$ )

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

7

8:25 / 9:31



### Lecture 2.3 - (Doc Length Normalization):

## What about Document Length?

Query = “news about presidential campaign”

d4

... news of **presidential campaign** ...  
... **presidential** candidate ...

**d6 > d4?**

**100 words**

d6

... **campaign** ..... **campaign** .....  
..... **news** .....  
.....  
..... **news** .....

.....  
.....  
.....  
.....

..... **presidential** ..... **presidential** .....

**5000 words**

1:47 / 18:56



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.

## Document Length Normalization

- Penalize a long doc with a doc length normalizer
  - Long doc has a better chance to match any query
  - Need to avoid over normalization

— NEED TO AVOID OVER-PENALIZATION

- A document is long because
  - it uses more words → more penalization
  - it has more contents → less penalization
- Pivoted length normalizer: average doc length as “pivot”
  - Normalizer = 1 if  $|d| = \text{average doc length (avdl)}$

4

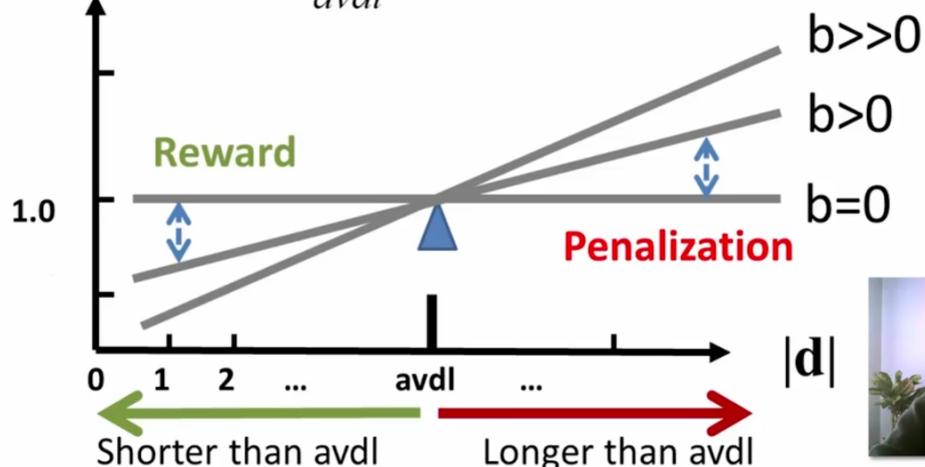


Although, documents can be long for different reasons. So, we need to take care whether document has more relevant or more non-relevant words.

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

### Pivoted Length Normalization

$$\text{normalizer} = 1 - b + b \frac{|d|}{\text{avdl}} \quad b \in [0, 1]$$



5



6:26 / 18:56

'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:

### State of the Art VSM Ranking Functions

- Pivoted Length Normalization VSM [Singhal et al 96]

$$f(q, d) = \sum_{w \in q \cap d} c(w, q) \frac{\ln[1 + \ln[1 + c(w, d)]]}{1 - b + b \frac{|d|}{avdl}} \log \frac{M + 1}{df(w)}$$

- BM25/Okapi [Robertson & Walker 94]

$$b \in [0, 1] \\ k_1, k_3 \in [0, +\infty)$$

$$f(q, d) = \sum_{w \in q \cap d} c(w, q) \frac{(k + 1)c(w, d)}{c(w, d) + k(1 - b + b \frac{|d|}{avdl})} \log \frac{M + 1}{df(w)}$$

6

▶ 8:48 / 18:56



Pivoted length - 'double log' TF transformation

BM25/Okapi - 'BM25' TF transformation

## Further Improvement of VSM?

- Improved instantiation of **dimension**?
  - stemmed words, stop word removal, phrases, latent semantic indexing (word clusters), character n-grams, ...
  - bag-of-words with phrases is often sufficient in practice
  - Language-specific and domain-specific tokenization is important to ensure “normalization of terms”
- Improved instantiation of **similarity function**?
  - cosine of angle between two vectors?
  - Euclidean?
  - dot product seems still the best (sufficiently general especially with appropriate term weighting)

7

▶ 12:27 / 18:56



### 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:**

## Summary of Vector Space Model

- $\text{Relevance}(q,d) = \text{similarity}(q,d)$
- Query and documents are represented as vectors
- Heuristic design of ranking function
- Major term weighting heuristics
  - TF weighting and transformation
  - IDF weighting
  - Document length normalization
- BM25 and Pivoted normalization seem to be most effective

9

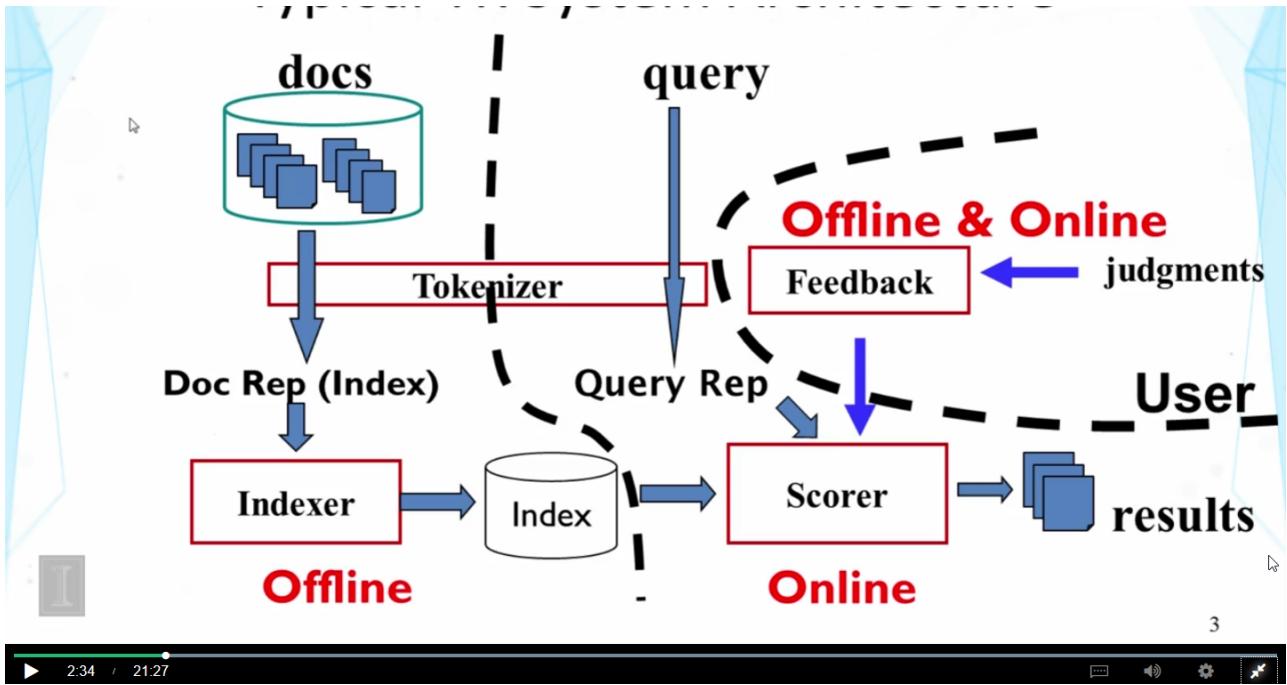
17:43 / 18:56



---

**Lecture 2.4 - (Implementation of TR systems):**

## Typical TR System Architecture



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

## Tokenization

- Normalize lexical units: Words with similar meanings should be mapped to the same indexing term
- Stemming: Mapping all inflectional forms of words to the same root form, e.g.
  - computer -> compute
  - computation -> compute
  - computing -> compute
- Some languages (e.g., Chinese) pose challenges in word segmentation



4

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

# Indexing

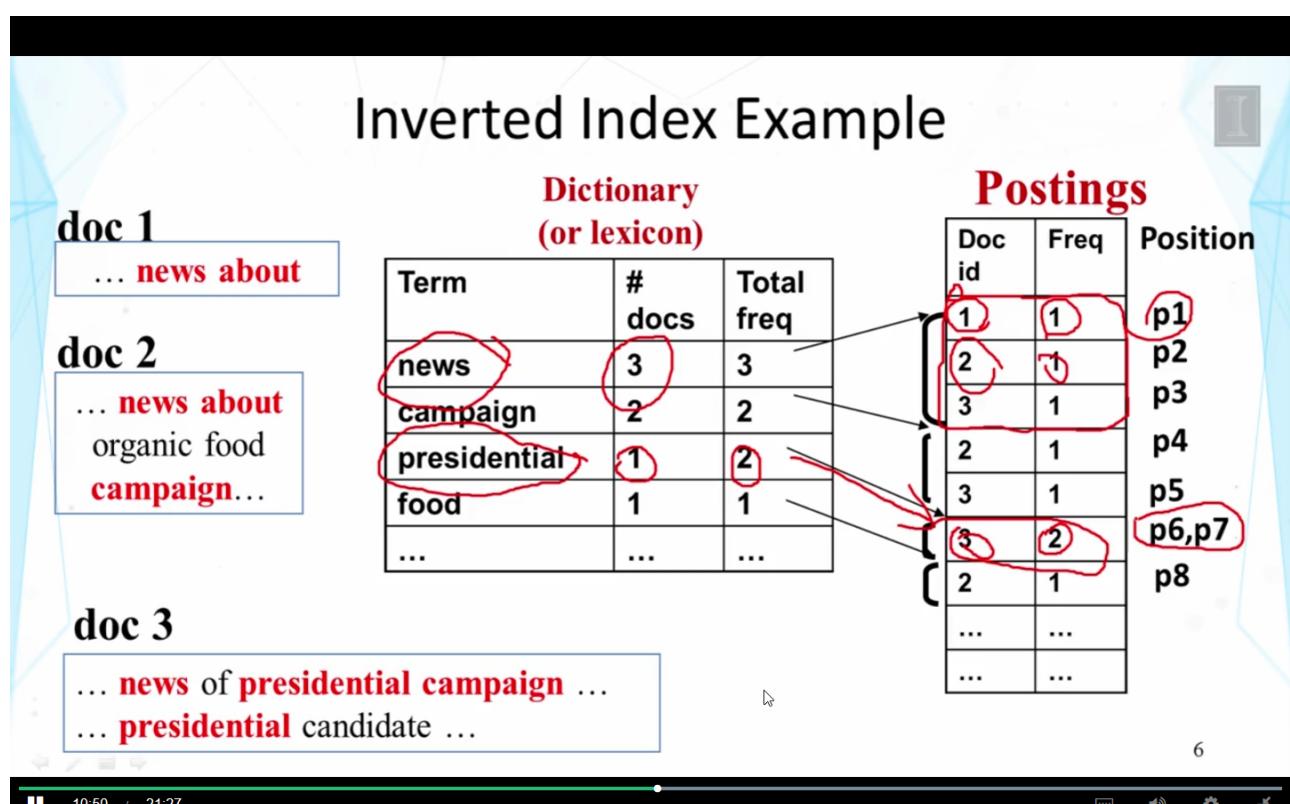
- Indexing = Convert documents to data structures that enable fast search (precomputing as much as we can)
- Inverted index is the dominating indexing method for supporting basic search algorithms
- Other indices (e.g., document index) may be needed for feedback



5

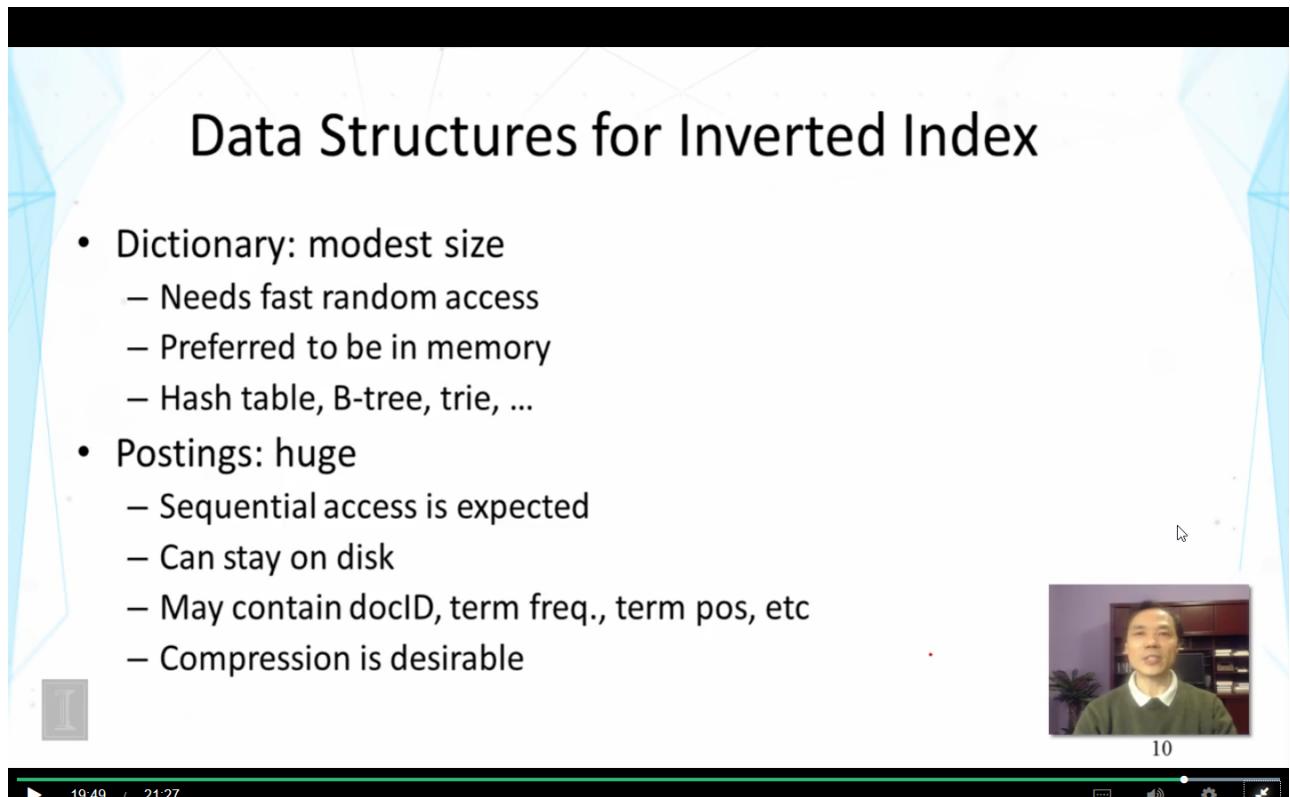
The query is also tokenized and then score is calculated. The user provides implicit or explicit feedback on the results.

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.



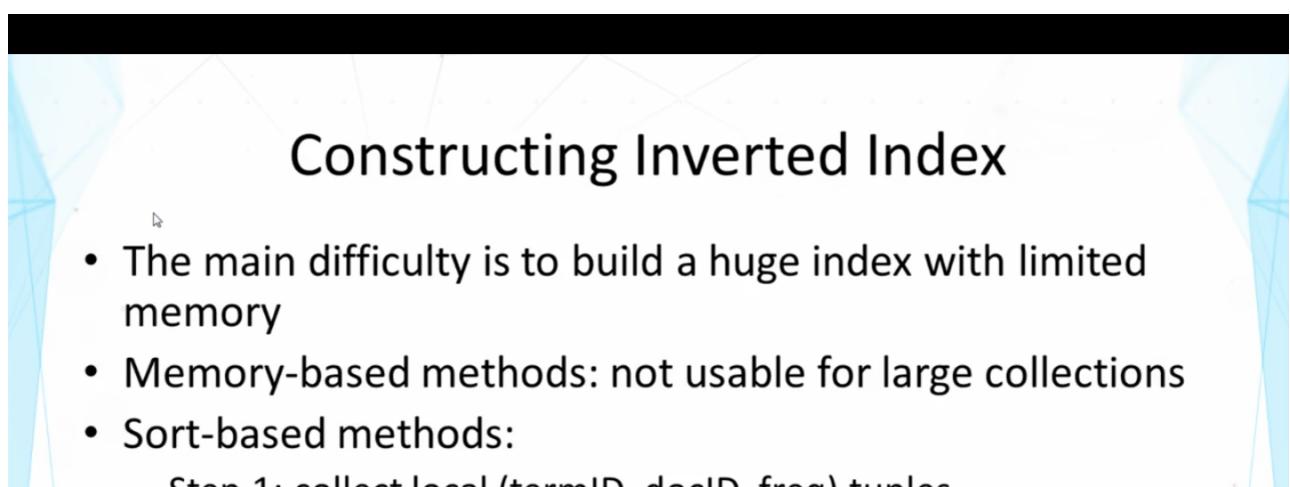
## Data Structures for Inverted Index

- Dictionary: modest size
  - Needs fast random access
  - Preferred to be in memory
  - Hash table, B-tree, trie, ...
- Postings: huge
  - Sequential access is expected
  - Can stay on disk
  - May contain docID, term freq., term pos, etc
  - Compression is desirable

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):



## Constructing Inverted Index

- The main difficulty is to build a huge index with limited memory
- Memory-based methods: not usable for large collections
- Sort-based methods:
  - Step 1: collect local (termID, docID, freq) tuples

- Step 1: collect local (termid, docid, freq) tuples
- Step 2: sort local tuples (to make “runs”)
- Step 3: pair-wise merge runs
- Step 4: Output inverted file

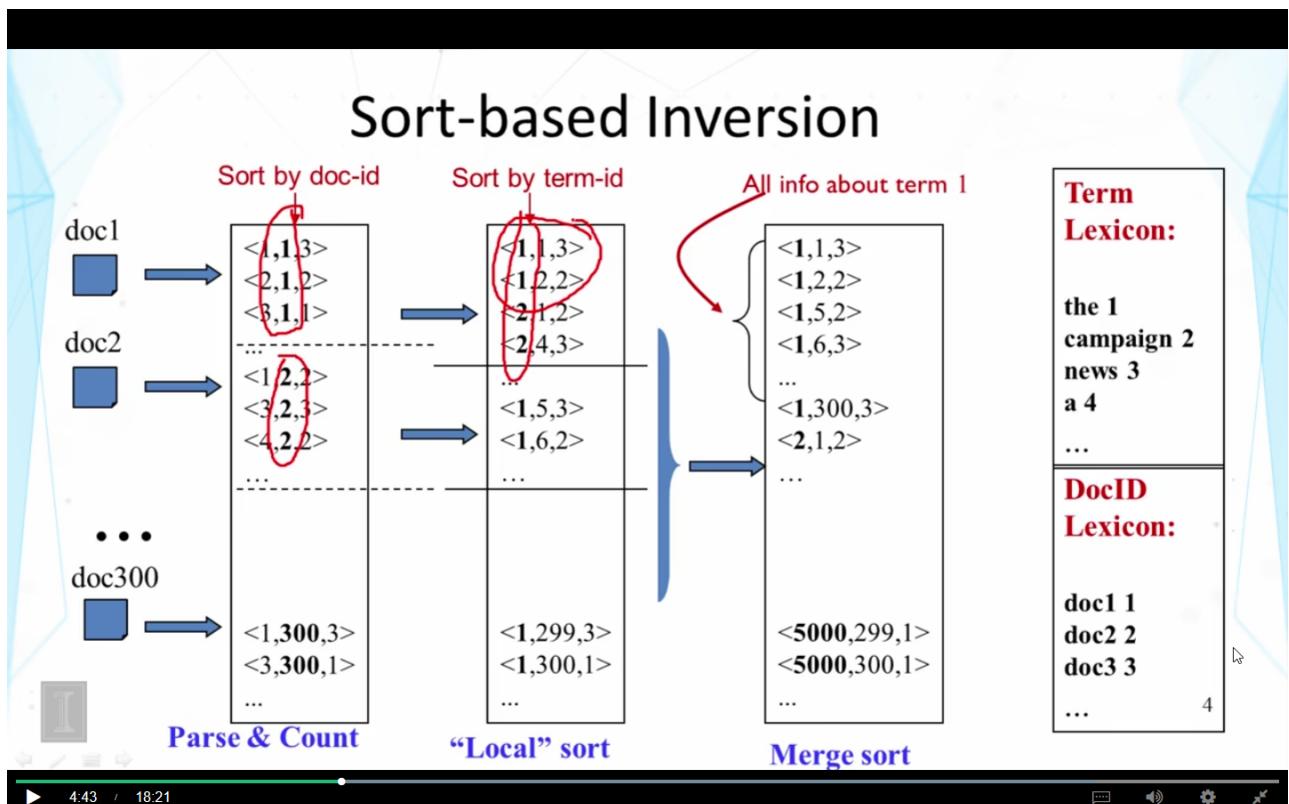


3

1:13 / 18:21



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.



4:43 / 18:21



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.

## Inverted Index Compression

- In general, leverage skewed distribution of values and use variable-length encoding
- TF compression
  - Small numbers tend to occur far more frequently than large numbers

(why?)

- Fewer bits for small (high frequency) integers at the cost of more bits for large integers
- Doc ID compression
  - “d-gap” (store difference):  $d_1, d_2-d_1, d_3-d_2, \dots$
  - Feasible due to sequential access



5

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.

## Integer Compression Methods

- Binary: equal-length coding
- Unary:  $x \geq 1$  is coded as  $x-1$  one bits followed by 0, e.g.,  
 $3 \Rightarrow 110; 5 \Rightarrow 11110$
- $\gamma$ -code:  $x \Rightarrow$  unary code for  $1 + \lfloor \log x \rfloor$  followed by uniform code for  $x - 2^{\lfloor \log x \rfloor}$  in  $\lfloor \log x \rfloor$  bits, e.g.,  $3 \Rightarrow 101, 5 \Rightarrow 11001$
- $\delta$ -code: same as  $\gamma$ -code ,but replace the unary prefix with  $\gamma$ -code. E.g.,  $3 \Rightarrow 1001, 5 \Rightarrow 10101$



6



Variable length encoding for integers

# Uncompress Inverted Index

- Decoding of encoded integers
  - Unary decoding: count 1's until seeing a zero
  - $\gamma$ -decoding
    - first decode the unary part; let value be  $k+1$
    - read  $k$  more bits decode them as binary code; let value be  $r$
    - the value of the encoded number is  $2^k+r$
- Decode doc IDs encoded using d-gap
  - Let the encoded ID list be  $x_1, x_2, x_3, \dots$
  - Decode  $x_1$  to obtain doc ID1; then decode  $x_2$  and add the recovered value to the doc ID1 just obtained
  - Repeatedly decode  $x_3, x_4, \dots$ , add the recovered value to the previous doc ID.

8

7



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

---

## Lecture 2.6 - (Fast Search):