Weeks 2

Goals and Objectives

After you actively engage in the learning experiences in this module, you should be able to:

* Explain what TF-IDF weighting is and why TF transformation and document length normalization are necessary for the design of an effective ranking function.
* TF-IDF 가중치가 무엇인지, 그리고 TF 변형과 문서 길이 정규화가 효과적인 순위 결정 기능을 설계하는 데 필요한 이유를 설명하십시오.
* Explain what an inverted index is and how to construct it for a large set of text documents that do not fit into the memory.
* 역 색인이 무엇인지 설명하고 메모리에 맞지 않는 대량의 텍스트 문서에 대해이를 어떻게 구성하는지 설명하십시오.
* Explain how variable-length encoding can be used to compress integers and how unary coding and gamma-coding work.
* 가변 길이 인코딩을 사용하여 정수를 압축하는 방법과 단항 코딩 및 감마 코딩이 어떻게 작동하는지 설명하십시오.
* Explain how scoring of documents in response to a query can be done quickly by using an inverted index.
* 거꾸로 된 색인을 사용하여 쿼리에 대한 응답으로 문서 채점을 빠르게 수행 할 수있는 방법을 설명하십시오.
* Explain Zipf’s law.
* 지프의 법칙을 설명하십시오.

Guiding Questions

Develop your answers to the following guiding questions while completing the readings and working on assignments throughout the week.

* What are some different ways to place a document as a vector in the vector space?
* 벡터 공간에서 문서를 벡터로 배치하는 몇 가지 다른 방법은 무엇입니까?
* What is term frequency (TF)?
* 용어 빈도 (TF) 란 무엇입니까?
* What is TF transformation?
* TF 변형이란 무엇입니까?
* What is document frequency (DF)?
* 문서 빈도 (DF) 란 무엇입니까?
* What is inverse document frequency (IDF)?
* IDF (inverse document frequency) 란 무엇입니까?
* What is TF-IDF weighting?
* TF-IDF 가중치 란 무엇입니까?
* Why do we need to penalize long documents in text retrieval?
* 텍스트 검색시 긴 문서에 벌점을 적용해야하는 이유는 무엇입니까?
* What is pivoted document length normalization?
* 피벗 된 문서 길이 정규화 란 무엇입니까?
* What are the main ideas behind the retrieval function BM25?
* 검색 기능 BM25의 주요 아이디어는 무엇입니까?
* What is the typical architecture of a text retrieval system?
* 텍스트 검색 시스템의 일반적인 아키텍처는 무엇입니까?
* What is an inverted index?
* 역 색인이란 무엇입니까?
* Why is it desirable to compress an inverted index?
* 거꾸로 된 인덱스를 압축하는 것이 왜 바람직한가요?
* How can we create an inverted index when the collection of documents does not fit into the memory?
* 문서 모음이 메모리에 맞지 않을 때 어떻게 역 색인을 만들 수 있습니까?
* How can we leverage an inverted index to score documents quickly?
* 역 색인을 활용하여 문서를 신속하게 점수화 할 수있는 방법은 무엇입니까?

Additional Readings and Resources

The following readings are optional:

* C. Zhai and S. Massung. *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining*, ACM Book Series, Morgan & Claypool Publishers, 2016. **Chapter 6 - Section 6.3, and Chapter 8.**
* Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second Edition. Morgan Kaufmann, 1999.

Key Phrases and Concepts

Keep your eyes open for the following key terms or phrases as you complete the readings and interact with the lectures. These topics will help you better understand the content in this module.

* Term frequency (TF)
* Document frequency (DF) and inverse document frequency (IDF)
* TF transformation
* Pivoted length normalization
* BM25
* Inverted index and postings
* Binary coding, unary coding, gamma-coding, and d-gap
* Zipf’s law

**2.1-TR-Vector\_Space\_Model\_Improve\_Instantiation**

**0:00**

[SOUND] This lecture is about how to do faster search by using invert index.

**0:14**

In this lecture, we're going to continue the discussion of system implementation. In particular, we're going to talk about how to support a faster search by using invert index.

**0:26**

So let's think about what a general scoring function might look like.

**0:32**

Now of course, the vector space model is a special case of this, but we can imagine many other retrieval functions of the same form.

**0:42**

So the form of this function is as follows.

**0:46**

We see this scoring function of a document D and a query Q is defined as first a function of fa that adjustment a function that would consider two factors. That I'll assume here at the end, f sub d of d and f sub q of q. These are adjustment factors of a document and a query, so they are at the level of a document and the query. So and then inside of this function, we also see there's another function called h. So this is the main part of the scoring function and these as I just said of the scoring factors at the level of the whole document and the query. For example, document [INAUDIBLE] and this aggregate punching would then combine all these. Now inside this h function, there are functions that would compute the weights of the contribution of a matched query term ti.

우리는 문서 D의이 채점 함수를보고 질의 Q는 두 가지 요소를 고려하는 함수를 조정하는 fa의 함수로 정의됩니다. 마지막에 여기에서 가정 할 것이고, f와 d의 하위 집합과 q의 집합입니다. 이것은 문서와 쿼리의 조정 요소이므로 문서와 쿼리의 수준에 있습니다. 그래서이 함수의 내부에는 h라는 또 다른 함수가 있습니다. 그래서 이것은 전체 문서와 쿼리 수준에서 채점하는 요소에 대해 말한 것처럼 채점 기능의 주요 부분입니다. 예를 들어 [INAUDIBLE] 문서와이 집계 펀치는이 모든 것을 결합합니다. 이제이 h 함수 내부에는 일치 된 쿼리 용어 ti의 기여도 가중치를 계산하는 함수가 있습니다.

**2:08**

So this g, the function g gives us the weight of a matched query term ti in document d.

그래서이 g, 함수 g는 문서 d에서 일치 된 질의어 ti의 가중치를줍니다.

**2:23**

And this h function would then aggregate all these weights. So for example, take a sum of all the matched query terms,

그리고이 h 함수는이 모든 가중치를 합산합니다. 예를 들어, 일치하는 모든 쿼리 용어의 합계를 취하고,

**2:36**

but it can also be a product or it could be another way of aggregating them.

제품이 될 수도 있고, 제품을 집계하는 다른 방법 일 수도 있습니다.

**2:41**

And then finally, this adjustment the functioning would then consider the document level or query level factors to further adjust this score, for example, document [INAUDIBLE]. So, this general form would cover many state of [INAUDIBLE] functions. Let's look at how we can score documents with such a function using virtual index.

그런 다음 마지막으로이 기능을 조정하면 문서 수준 또는 쿼리 수준 요소를 고려하여이 점수를 추가로 조정할 수 있습니다 (예 : document [INAUDIBLE]). 그래서,이 일반적인 형식은 [INAUDIBLE] 기능의 많은 상태를 다룰 것입니다. 가상 인덱스를 사용하여 이러한 함수로 문서를 채점하는 방법을 살펴 보겠습니다.

**3:07**

So, here's a general algorithm that works as follows. First this query level and document level factors can be pre-computed in the indexing time. Of course, for the query we have to compute it at the query time but for document, for example, document [INAUDIBLE] can be pre-computed. And then, we maintain a score accumulator for each document d to computer h.

여기에 다음과 같이 작동하는 일반적인 알고리즘이 있습니다. 먼저이 쿼리 수준 및 문서 수준 요소는 인덱싱 시간에 미리 계산 될 수 있습니다. 물론 쿼리의 경우 쿼리 시간에 계산해야하지만 문서의 경우 [INAUDIBLE] 문서를 미리 계산할 수 있습니다. 그리고 나서, 우리는 컴퓨터에 각 문서 d에 대한 점수 누적기를 유지합니다.

**3:34**

An h is an aggregation function over all the matching query terms. So how do we do that? For each period term we're going to do fetch the inverted list from the invert index. This will give us all the documents that match this query term

**3:52**

and that includes d1, f1 and so dn fn. So each pair is a document ID and the frequency of the term in the document. Then for each entry d sub j and f sub j are particular match of the term in this particular document d sub j. We'll going to compute the function g that would give us something like weight of this term, so we're computing the weight completion of matching this query term in this document. And then, we're going to update the score accumulator for this document and this would allow us to add this to our accumulator that would incrementally compute function h. So this is basically a general way to allow pseudo computer or functions of this form by using the inbound index. Note that we don't have to attach any of document and that didn't match any query term. Well, this is why it's fast, we only need to process the documents that matched at least one query term. In the end, then we're going to adjust the score the computer this function f sub a and then we can sort. So let's take a look at a specific example. In this case, let's assume the scoring function is a very simple one, it just takes the sum of t f, the role of t f, the count of a term in the document.

h는 모든 일치하는 u 리 용어에 대한 집계 함수입니다. 그럼 우리가 어떻게해야합니까? 각 기간에 대해 역 색인에서 반전 된 목록을 가져 오려고합니다. 이렇게하면이 검색어와 일치하고 d1, f1 및 dn fn을 포함하는 모든 문서가 제공됩니다. 따라서 각 쌍은 문서 ID 및 문서의 용어 빈도입니다. 그런 다음 각 항목에 대해 d sub j 및 f sub j는이 특정 문서 d sub j에서 용어의 특정 일치입니다. 우리는이 용어의 가중치와 같은 것을주는 함수 g를 계산할 것이므로이 문서에서이 쿼리 용어와 일치하는 가중치 완성을 계산할 것입니다. 그리고 나서이 문서의 점수 누적기를 업데이트 할 것입니다. 그러면 누적 계산기에이를 추가하여 점진적으로 함수 h를 계산할 수 있습니다. 따라서 기본적으로 인바운드 인덱스를 사용하여 의사 (pseudo) 컴퓨터 또는이 양식의 기능을 허용하는 일반적인 방법입니다. 문서를 첨부 할 필요가 없으며 검색어가 일치하지 않습니다. 글쎄, 이것이 빠른 이유이기 때문에 적어도 하나의 쿼리 용어와 일치하는 문서 만 처리하면됩니다. 결과적으로 컴퓨터에서이 점수를 조정하고 점수를 매겨서 정렬 할 수 있습니다. 이제 구체적인 예를 살펴 보겠습니다. 이 경우 득점 함수가 매우 단순하다고 가정 해 보겠습니다.이 함수는 t f의 합계, t f의 역할, 문서의 용어 수를 취합니다.

**5:25**

This simplification would help shield the algorithm clearly. It's very easy to extend the computation to include other weights like the transformation of tf, or [INAUDIBLE] or IDF [INAUDIBLE]. So let's take a look at specific example, where the queries information security

**5:48**

and it show some entries of invert index on the right side. Information occurred in four documents and their frequencies are also there, security occurred in three documents. So let's see how the arrows works, so first we iterate overall query terms and we fetch the first query then, what is that? That's information, right? And imagine we have all these score accumulators who score the,

**6:17**

scores for these documents. We can imagine there will be other but then they will only be allocated as needed. So before we do any waiting of terms, we don't even need a score of. That comes actually we have these score accumulators eventually allocating.

**6:38**

So lets fetch the interest from the entity [INAUDIBLE] for information, that the first one.

**6:46**

So these four accumulators obviously would be initialize as zeros.

이 단순화는 알고리즘을 명확하게 보호하는 데 도움이됩니다. tf 또는 [INAUDIBLE] 또는 IDF [INAUDIBLE]의 변환과 같은 다른 가중치를 포함하도록 계산을 확장하는 것은 매우 쉽습니다. 이제 특정 예제를 살펴 보겠습니다. 여기서 쿼리 정보 보안과 오른쪽에 반전 인덱스 항목이 표시됩니다. 정보는 네 개의 문서에서 발생했으며 그 빈도도 있으며 세 가지 문서에서 보안이 발생했습니다. 이제 화살표가 어떻게 작동하는지 살펴 보겠습니다. 먼저 전체 쿼리 용어를 반복하고 첫 번째 쿼리를 가져옵니다. 그게 무엇입니까? 그게 정보 야, 맞지? 그리고이 모든 점수 누적 점수를 가진 점수 기록자가 있다고 가정 해보십시오. 우리는 다른 사람들이있을 것이라고 상상할 수 있지만, 필요할 때만 배정 될 것입니다. 그래서 우리가 조건을 기다리기 전에, 우리는 심지어 점수를 필요로하지 않습니다. 사실 우리는 결국 점수 누적기를 결국 할당하게됩니다. 따라서 엔티티 [INAUDIBLE]가 첫 번째 정보에 대한 관심을 가져올 수 있습니다. 따라서이 4 개의 누산기는 분명히 0으로 초기화됩니다.

**6:51**

So, the first entry is d1 and 3, 3 is occurrences of information in this document. Since our scoring function assume that the score is just a sum of these raw counts. We just need to add a 3 to the score accumulator to account for the increase of score due to matching this term information, a document d1. And then, we go to the next entry, that's d2 and 4 and then we add a 4 to the score accumulator of d2. Of course, at this point, that we will allocate the score accumulator as needed. And so at this point, we allocated the d1 and d2, and the next one is d3, and we add one, we allocate another score [INAUDIBLE] d3 and add one to it. And then finally, the d4 gets a 5, because the term information occurred five times in this document. Okay, so this completes the processing of all the entries in the invert index for information. It processed all the contributions of matching information in this four documents.

따라서 첫 번째 항목은 d1과 3이며, 3은이 문서에서 정보가 표시됩니다. 우리의 채점 함수는 점수가이 날 카운트의 합계라고 가정하기 때문에. 이 용어 정보와 일치하는 점수의 증가를 설명하기 위해 점수 누적 계산기에 3을 추가하면됩니다. 문서 d1. 그런 다음 d2와 4의 다음 항목으로 이동 한 다음 d2의 점수 누적기에 4를 더합니다. 물론,이 시점에서 우리는 필요에 따라 점수 누적기를 할당 할 것입니다. 그래서이 시점에서 d1과 d2를 할당하고 다음 d3을 할당하고 하나를 더하고 다른 점수 [INAUDIBLE] d3을 할당하고 하나를 더합니다. 마지막으로 d4는 5라는 정보를 얻습니다. 왜냐하면 용어 정보가이 문서에서 5 번 발생했기 때문입니다. 좋아, 이렇게하면 정보에 대한 역 색인의 모든 항목 처리가 완료됩니다. 이 네 가지 문서에서 일치하는 정보의 모든 기여를 처리했습니다.

**8:01**

So now, our error will go to the next that's security. So, we're going to fetch all the inverted index entries for security.

이제는 오류가 보안 문제로 이어질 것입니다. 따라서 우리는 보안을 위해 모든 역 색인 항목을 가져올 것입니다.

**8:10**

So, in this case, there are three entries, and we're going to go through each of them. The first is d2 and 3 and that means security occur three humps in d2 and what do we do? Well, we do exactly the same, as what we did for information. So, this time we're going to change the score [INAUDIBLE] d2 since it's already allocated and what we do is we'll add 3 to the existing value which is a 4, so we now get a 7 for d2.

따라서이 경우 세 개의 항목이 있으며 각 항목을 살펴 보겠습니다. 첫 번째는 d2와 3이며 이는 보안이 d2에서 세 개의 혹이 발생한다는 것을 의미하며 우리는 무엇을합니까? 우리가 정보를 위해했던 것과 똑같은 일을합니다. 그래서,이 시간 우리는 이미 할당 이후 점수 [들리지] (D2)를 변경하려고하고 우리가 할 것은 우리가 4 기존 값에 3을 추가 할 것입니다, 그래서 우리는 지금 (D2)에 대한 7을 얻을 수 있습니다.

**8:41**

D2 score is increased because the match that falls the information and the security. Go to the next entry, that's d4 and 1, so we would the score for d4 and again, we add 1 to d4 so d4 goes from 5 to 6. Finally, we process d5 and a 3. Since we have not yet allocated a score accumulated for d5, at this point, we're going to allocate 1 for d5, and we're going to add a 3 to it. So, those scores, of the last rule, are the final scores for these documents.

D2 점수는 정보와 보안을 떨어 뜨리기 때문에 증가합니다. 다음 항목으로 이동하면 d4와 1이므로 d4에 대한 점수를 얻고 다시 d4를 1로 추가하여 d4가 5에서 6으로 변경됩니다. 마지막으로 d5와 a3을 처리합니다. d5에 대해 누적 된 점수는 d5에 1을 할당하고 여기에 3을 더합니다. 따라서 마지막 규칙의 점수는이 문서의 최종 점수입니다.

**9:20**

If our scoring function is just a simple some of TF values.

**9:27**

Now, what if we, actually, would like to do form addition? Well, we going to do the [INAUDIBLE] at this point, for each document.

**9:36**

So, to summarize this, all right, so you can see, we first process the information determine query term information and we processed all the entries in what index for this term. Then we process the security, all right, its worst think about what should be the order of processing here when we can see the query terms? It might make a difference especially if we don't want to keep all the score accumulators. Let's say, we only want to keep the most promising score accumulators. What do you think would be a good order to go through? Would you process a common term first or would you process a rare term first?

우리의 채점 기능이 TF 값 중 일부일뿐입니다. 자, 실제로 양식을 추가하고 싶다면 어떻게해야할까요? 우리는이 시점에서 각 문서에 대해 [무관심]을 수행 할 것입니다. 그래서 이것을 요약하면 알 수 있듯이 정보를 쿼리 정보 용어로 처리하고이 용어에 대한 색인의 모든 항목을 처리했습니다. 그런 다음 보안을 처리합니다. 좋습니다. 최악의 경우 쿼리 순서를 볼 때 처리 순서가 무엇인지 생각해보십시오. 우리가 모든 점수 누적기를 유지하고 싶지 않다면 특히 효과가있을 것입니다. 예를 들어, 우리는 가장 유망한 점수 누적기를 유지하기를 원합니다. 통과하기에 좋은 순서라고 생각하니? 먼저 공통 용어를 처리 하시겠습니까? 아니면 희귀 용어를 먼저 처리 하시겠습니까?

**10:24**

The answers is we just go to who should process the rare term first. A rare term would match a few documents, and then the score contribution would be higher, because the ideal value would be higher. And then, it allows us to attach the most diplomacy documents first. So, it helps pruning some non-promising ones, if we don't need so many documents to be returned to the user. So those are all heuristics for further improving the accuracy. Here you can also see how we can incorporate the idea of waiting, right? So they can [INAUDIBLE] when we incorporate [INAUDIBLE] when we process each query time. When we fetch the inverted index we can fetch the document frequency and then we can compute IDF. Or maybe perhaps the IDF value has already been precomputed when we indexed the documents. At that time, we already computed the IDF value that we can just fetch it, so all these can be done at this time. So that would mean when we process all the entries for information, these words would be adjusted by the same IDF, which is IDF for information.

대답은 드문 용어를 먼저 처리해야하는 사람에게로 이동하는 것입니다. 드문 용어는 몇 가지 문서와 일치 할 것이고 이상적인 값이 더 높기 때문에 점수 기여도가 더 높을 것입니다. 그리고 가장 먼저 외교 문서를 첨부 할 수 있습니다. 그래서 우리가 너무 많은 문서가 사용자에게 반환 될 필요가 없다면, 일부 유망하지 않은 것들을 가지 치기 (pruning) 도움이됩니다. 그래서 그것들은 정확성을 더욱 향상시키기위한 발견 적 방법입니다. 여기서 우리는 대기의 아이디어를 어떻게 통합 할 수 있는지도 볼 수 있습니다. 맞습니까? 따라서 우리는 각 질의 시간을 처리 할 때 [INAUDIBLE]을 통합 할 때 [INAUDIBLE] 할 수 있습니다. 역 색인을 가져올 때 문서 빈도를 가져온 다음 IDF를 계산할 수 있습니다. 아니면 우리가 문서를 색인 할 때 이미 IDF 값이 미리 계산되어있을 수도 있습니다. 그 당시 우리는 이미 가져올 수있는 IDF 값을 이미 계산 했으므로이 모든 것이 현재 수행 될 수 있습니다. 따라서 모든 정보 항목을 처리 할 때 동일한 단어 IDF (정보 IDF)가이 단어를 조정하게됩니다.

**11:36**

So this is the basic idea of using inverted index for fast research and it works well for all kinds of formulas that are of the general form. And this generally, the general form covers actually most state of art retrieval functions. So there are some tricks to further improve the efficiency, some general techniques to encode the caching. This is we just store some results of popular queries, so that next time when you see the same query, you simply return the stored results. Similarly, you can also slow the list of inverted index in the memory for a popular term. And if the query term is popular likely, you will soon need to factor the inverted index for the same term again. So keeping it in the memory would help, and these are general techniques for improving efficiency. We can also keep only the most promising accumulators because a user generally doesn't want to examine so many documents. We only need to return high qualities subset of documents that likely are ranked on the top.

따라서 이것은 빠른 연구를 위해 역 색인을 사용하는 기본적인 아이디어이며 일반적인 형태의 모든 수식에 적합합니다. 그리고 이것은 일반적으로, 일반적인 형태는 사실 예술 검색 기능의 대부분의 상태를 포함합니다. 그래서 효율성을 향상시키기위한 몇 가지 트릭, 캐싱을 인코딩하는 몇 가지 일반적인 기술이 있습니다. 이것은 인기있는 쿼리 결과를 저장하기 때문에 다음에 같은 쿼리를 볼 때 저장된 결과 만 반환하면됩니다. 비슷하게 인기있는 용어의 경우 메모리에서 역 색인의 목록을 느리게 할 수도 있습니다. 그리고 검색어가 인기있는 경우 곧 같은 용어에 대한 역 색인을 다시 고려해야합니다. 따라서 그것을 기억에 유지하는 것이 도움이 될 것이며, 이는 효율성을 향상시키는 일반적인 기술입니다. 사용자가 일반적으로 너무 많은 문서를 검토하기를 원하지 않기 때문에 가장 유망한 축전기 만 유지할 수도 있습니다. 최상위 순위에 올 가능성이 높은 문서의 하위 집합만을 반환해야합니다.

**12:47**

For that purpose, we can then prune the accumulators. We don't have to store all the accumulators. At some point, we just keep the highest value accumulators. Another technique is to do parallel processing and that's needed for really process in such a large data set like the web data set. And you scale up to the Web-scale really to have the special techniques you do parallel processing and to distribute the storage of files on machines. So here is a list of some text retrieval toolkits, it's not a complete list. You can find more information at this URL on the bottom. And here, I listed your four here, Lucene's one of the most popular toolkits that can support a lot of applications and it has very nice support for applications. You can use it to build a search engine application very quickly. The downside is that it's not that easy to extend it, and the algorithms implemented they are also not the most advanced algorithms. Lemur or Indri is another toolkit that does not have such a nice support web application as Lucene but it has many advanced search algorithms and it's also easy to extend. Terrier is yet another toolkit that also has good support for application capability and some advanced algorithms. So that's maybe in between Lemur or Lucene, or maybe rather combining the strands of both, so that's also useful tool kit. MeTA is a toolkit that we will use for the problem assignment and this is a new toolkit that has

**14:47**

a combination of both text retrieval algorithms and text mining algorithms. And so talking models are implement they are a number of text analysis algorithms implemented in the toolkit as well as basic search algorithms. So to summarize all the discussion about the System Implementation,

**15:11**

here are the major takeaway points. Inverted index is the primary data structure for supporting a search engine and that's the key to enable faster response to a user's query.

그 목적을 위해, 우리는 누적기를 정리할 수 있습니다. 우리는 모든 축전지를 저장할 필요가 없습니다. 어떤 시점에서, 우리는 단지 가장 높은 값의 누적기를 유지합니다. 또 다른 기술은 병렬 처리를 수행하는 것이며 웹 데이터 세트와 같은 대규모 데이터 세트에서 실제로 처리하는 데 필요합니다. 또한 웹 스케일로 확장하여 병렬 처리를 수행하고 컴퓨터에 파일 저장 영역을 분산시키는 특별한 기술을 보유하게됩니다. 여기 몇몇 텍스트 검색 툴킷 목록이 있습니다. 완전한 목록은 아닙니다. 자세한 내용은 하단의이 URL을 참조하십시오. 여기 Lucene의 많은 네 가지 애플리케이션을 지원할 수있는 가장 인기있는 툴킷 중 하나 인 네 가지를 나열 했으므로 애플리케이션에 대한 지원이 매우 좋습니다. 이 도구를 사용하여 매우 빠르게 검색 엔진 응용 프로그램을 만들 수 있습니다. 단점은 확장하기가 쉽지 않으며 구현 된 알고리즘이 가장 진보 된 알고리즘이 아니라는 것입니다. Lemur 또는 Indri는 Lucene과 같은 훌륭한 지원 웹 응용 프로그램이없는 또 다른 툴킷이지만 많은 고급 검색 알고리즘이 있으며 확장하기도 쉽습니다. Terrier는 응용 프로그램 기능 및 일부 고급 알고리즘을 잘 지원하는 또 다른 툴킷입니다. 그래서 아마 Lemur 또는 Lucene 사이에있을 수도 있고, 아니면 양쪽 모두의 가닥을 결합하는 것일 수도 있으므로 유용한 도구 키트이기도합니다. MeTA는 문제 할당을 위해 사용할 툴킷이며, 텍스트 추출 알고리즘과 텍스트 마이닝 알고리즘을 결합한 새로운 툴킷입니다. 그리고 말하는 모델은 툴킷과 기본 검색 알고리즘에서 구현되는 수많은 텍스트 분석 알고리즘입니다. 따라서 시스템 구현에 대한 모든 논의를 요약하면 다음과 같습니다. 반전 된 색인은 검색 엔진을 지원하기위한 기본 데이터 구조이며 사용자 쿼리에보다 신속하게 응답 할 수있는 열쇠입니다.

**15:26**

And the basic idea is to preprocess the data as much as we can, and we want to do compression when appropriate. So that we can save disk space and we can speed up IO and processing of inverted index in general. We talked about how to construct the invert index when the data can't fit into the memory. And then we talk about faster search using that index basically, what's we exploit the invective index to accumulate a scores for documents [INAUDIBLE] algorithm. And we exploit the Zipf's law to avoid the touching many documents that don't match any query term and this algorithm can actually support a wide range of ranking algorithms.

그리고 기본 아이디어는 가능한 한 많은 양의 데이터를 전처리하는 것이며 적절한 경우 압축을 수행하려고합니다. 그래서 우리는 디스크 공간을 절약 할 수 있고 전반적인 역 색인의 IO와 처리 속도를 높일 수 있습니다. 데이터가 메모리에 맞지 않을 때 역 색인을 만드는 방법에 대해 이야기했습니다. 그런 다음 기본적으로 색인을 사용하는 빠른 검색에 대해 이야기합니다. 문서의 [INAUDIBLE] 알고리즘 점수를 누적하는 비 활동 색인을 활용하는 것은 무엇입니까? 그리고 우리는 쿼리 용어와 일치하지 않는 많은 문서를 만지는 것을 피하기 위해 지프의 법칙을 이용합니다.이 알고리즘은 실제로 광범위한 순위 알고리즘을 지원할 수 있습니다.

**16:13**

So these basic techniques have great potential for further scaling up using distributed file system, parallel processing, and caching. Here are two additional readings you can take a look, if you have time and you are interested in learning more about this. The first one is a classical textbook on the efficiency o inverted index and the compression techniques. And how to, in general feel that the efficient any inputs of the space, overhead and speed. The second one is a newer textbook that has a nice discussion of implementing and evaluating search engines.

따라서 이러한 기본 기술은 분산 파일 시스템, 병렬 처리 및 캐싱을 사용하여 추가 확장이 가능합니다. 시간이 있고 이것에 대해 더 많은 것을 배우고 싶다면 두 가지 더 읽을 수 있습니다. 첫 번째는 역 색인 및 압축 기술의 효율성에 대한 고전 교과서입니다. 그리고 어떻게, 일반적으로 공간, 오버 헤드 및 속도의 효율적인 입력. 두 번째는 검색 엔진을 구현하고 평가하는 좋은 토론이있는 최신 교과서입니다.

**2.2-TR-TF\_Transformation**

**0:10**

In this lecture, we continue the discussion of vector space model. In particular, we're going to talk about the TF transformation. In the previous lecture, we have derived a TF idea of weighting formula using the vector space model.

이 강의에서는 벡터 공간 모델에 대한 논의를 계속합니다. 특히 우리는 TF 변형에 대해 이야기 할 것입니다. 이전 강연에서는 벡터 공간 모델을 사용하여 가중치 공식의 TF 아이디어를 도출했습니다.

**0:27**

And we have assumed that this model actually works pretty well for these examples as shown on this slide, except for d5, which has received a very high score. Indeed, it has received the highest score among all these documents. But this document is intuitive and non-relevant, so this is not desirable.

그리고 우리는이 슬라이드가 d5를 제외하고 매우 높은 점수를 얻은 것을 제외하면이 모델이 실제로이 예제에서 꽤 잘 작동한다고 가정했습니다. 실제로이 모든 문서 중에서 가장 높은 점수를 받았습니다. 그러나이 문서는 직관적이고 관련성이 없으므로 바람직하지 않습니다.

**0:53**

In this lecture, we're going to talk about, how we're going to use TF transformation to solve this problem.

**1:00**

Before we discuss the details, let's take a look at the formula for this simple TF-IDF weighting ranking function. And see why this document has received such a high score. So this is the formula, and if you look at the formula carefully, then you will see it involves a sum over all the matched query terms.

**1:23**

And inside the sum, each matched query term has a particular weight. And this weight is TF-IDF weighting.

**1:31**

So it has an idea of component, where we see two variables. One is the total number of documents in the collection, and that is M. The other is the document of frequency. This is the number of documents that are contained. This word w. The other variables involved in the formula include the count of the query term.

**2:01**

W in the query, and the count of the word in the document.

세부 사항을 논의하기 전에이 간단한 TF-IDF 가중치 지정 함수의 수식을 살펴 보겠습니다. 그리고 왜이 문서가 그렇게 높은 점수를 받았는지보십시오. 따라서 이것은 수식입니다. 수식을 신중하게 보면 일치하는 모든 검색어에 대해 합계가 표시됩니다. 그리고 합계 내에서 일치하는 각 검색어는 특정 가중치를가집니다. 그리고이 무게는 TF-IDF 가중치입니다. 그래서 두 개의 변수를 볼 수있는 구성 요소에 대한 아이디어가 있습니다. 하나는 컬렉션에있는 문서의 총 개수이며, M입니다. 다른 하나는 빈도 문서입니다. 이것은 포함 된. 서 수입니다. 이 단어 w. 수식에 포함 된 다른 변수에는 쿼리 용어의 개수가 포함됩니다. W는 쿼리에서, 그리고 단어의 카운트는 문서에서.

**2:07**

If you look at this document again, now it's not hard to realize that the reason why it hasn't received a high score is because it has a very high count of campaign. So the count of campaign in this document is a 4, which is much higher than the other documents, and has contributed to the high score of this document. So in treating the amount to lower the score for this document, we need to somehow restrict the contribution of the matching of this term in the document. And if you think about the matching of terms in the document carefully, you actually would realize, we probably shouldn't reward multiple occurrences so generously. And by that I mean, the first occurrence of a term says a lot about the matching of this term, because it goes from zero count to a count of one. And that increase means a lot.

이 문서를 다시 보면 지금은 높은 점수를받지 못한 이유는 캠페인 수가 매우 높기 때문이라는 것을 알기가 어렵지 않습니다. 따라서이 문서의 캠페인 수는 다른 문서보다 훨씬 많은 4이며이 문서의 높은 점수에 기여했습니다. 따라서이 문서의 점수를 낮추기위한 금액을 처리 할 때 우리는 어떻게해서 문서에서이 용어의 일치의 기여도를 제한해야합니다. 그리고 문서의 용어를 신중하게 일치시키는 것에 대해 생각해 보면 실제로 여러 번 발생하는 것을 충분히 관대하게 보지 않아야합니다. 그리고 그것이 의미하는 바로는, 용어의 첫 번째 발생은이 용어의 일치에 대해 많이 말합니다. 왜냐하면 그것은 0부터 1까지의 숫자를 가지기 때문입니다. 그리고 그 증가는 많은 것을 의미합니다.

**3:17**

Once we see a word in the document, it's very likely that the document is talking about this word. If we see a extra occurrence on top of the first occurrence, that is to go from one to two, then we also can say that, well the second occurrence kind of confirmed that it's not a accidental managing of the word. Now we are more sure that this document is talking about this word. But imagine we have seen, let's say, 50 times of the word in the document. Now, adding one extra occurrence is not going to test more about the evidence, because we're already sure that this document is about this word.

문서에서 한 단어를 발견하면이 단어에 대한 문서가 작성 될 가능성이 큽니다. 우리가 첫 번째 사건의 꼭대기에 추가 사건을 보게된다면 그것은 하나에서 두 가지로 진행될 것입니다. 그러면 우연한 두 번째 사건이 단어의 우발적 인 관리가 아니라고 확신 할 수 있습니다. 이제이 문서가이 단어에 대해 이야기하고 있음을 확신합니다. 그러나 문서에서 단어의 50 배를 보았다고 가정 해 봅시다. 이제 증거를 더 많이 테스트하지 않을 것입니다. 왜냐하면 우리는 이미이 문서가이 단어에 관한 것이라고 확신하기 때문입니다.

**4:01**

So if you're thinking this way, it seems that we should restrict the contribution of a high count of a term, and that is the idea of TF Transformation. So this transformation function is going to turn the real count of word into a term frequency weight for the word in the document. So here I show in x axis that we'll count, and y axis I show the term frequency weight.

그래서 당신이 이런 식으로 생각한다면, 우리는 용어의 높은 숫자의 기여를 제한해야하는 것으로 보입니다. 그것은 TF 변형의 아이디어입니다. 따라서이 변환 함수는 실제 단어 수를 문서의 단어에 대한 용어 빈도 가중치로 변환합니다. 여기서는 x 축에 세고, y 축에 주파수 체중을 표시합니다.

**4:33**

So in the previous breaking functions, we actually have imprison rate use some kind of transformation. So for example, in the 0/1 bit vector recantation,

**4:44**

we actually use such a transformation function, as shown here. Basically if the count is 0, then it has 0 weight, otherwise it would have a weight of 1. It's flat.

**4:59**

Now, what about using term count as TF weight? Well, that's a linear function, so it has just exactly the same weight as the count.

**5:11**

Now we have just seen that this is not desirable.

**5:18**

So what we want is something like this. So for example, with an algorithm function, we can't have a sublinear transformation that looks like this. And this will control the influence of really high weight, because it's going to lower its inference. Yet, it will retain the inference of small counts.

**5:36**

Or we might want to even bend the curve more by applying logarithm twice.

따라서 이전의 위반 기능에서 우리는 사실 투옥 율을 사용하여 일종의 변형을 사용합니다. 예를 들어, 0/1 비트 벡터 재귀에서, 우리는 실제로 여기에 표시된 것과 같은 변형 함수를 사용합니다. 기본적으로 카운트가 0이면 0이되고, 그렇지 않으면 1의 가중치를가집니다. 이제 TF 가중치로 용어 집계를 사용하는 것은 어떻습니까? 음, 선형 함수입니다. 따라서 정확히 같은 가중치를가집니다. 이제 우리는 이것이 바람직하지 않다는 것을 보았습니다. 그래서 우리가 원하는 것은 이런 것입니다. 예를 들어 알고리즘 함수를 사용하면 다음과 같은 서브 라인 변형을 사용할 수 없습니다. 그리고 이것은 정말로 높은 무게의 영향력을 통제 할 것입니다. 왜냐하면 그것이 그 추론을 낮출 것이기 때문입니다. 그러나, 그것은 작은 숫자의 추론을 유지합니다. 또는 로그를 두 번 적용하여 곡선을 더 구부릴 수도 있습니다.

**5:42**

Now people have tried all these methods. And they are indeed working better than the linear form of the transformation.

**5:50**

But so far, what works the best seems to be this special transformation, called a BM25 transformation.

**5:58**

BM stands for best matching.

이제 사람들은이 모든 방법을 시도했습니다. 그리고 그들은 실제로 변형의 선형 형태보다 잘 작동합니다. 그러나 지금까지 가장 잘 작동하는 것은 BM25 변환이라고 불리는이 특별한 변환 인 것 같습니다. BM은 가장 잘 어울리는 것을 의미합니다.

**6:01**

Now in this transformation, you can see there's a parameter k here.

**6:06**

And this k controls the upper bound of this function. It's easy to see this function has a upper bound, because if you look at the x divided by x + k, where k is a non-active number, then the numerator will never be able to exceed the denominator, right? So it's upper bounded by k+1. Now, this is also difference between this transformation function and a logarithm transformation.

**6:37**

Which it doesn't have upper bound.

**6:39**

Furthermore, one interesting property of this function is that, as we vary k,

**6:45**

we can actually simulate different transformation functions. Including the two extremes that are shown here. That is, the 0/1 bit transformation and the linear transformation. So for example, if we set k to 0, now you can see

**7:03**

the function value will be 1. So we precisely recover the 0/1 bit transformation.

이제 이 변환에서 매개 변수 k가 있음을 알 수 있습니다. 그리고이 k는이 함수의 상한을 제어합니다. 이 함수가 상한을 갖는 것을 쉽게 알 수 있습니다. 왜냐하면 x + k로 나눈 x를 보았을 때, k는 비활성 숫자이고 분자는 분모를 초과 할 수 없기 때문입니다. 그래서 k + 1에 의해 상한이됩니다. 자, 이것은이 변환 함수와 대수 변환의 차이점이기도합니다. 그것은 상한이 없습니다. 또한,이 함수의 흥미로운 특성 중 하나는 k를 변화 시킴에 따라 서로 다른 변환 함수를 실제로 시뮬레이션 할 수 있다는 것입니다. 여기에 표시된 두 극단을 포함합니다. 즉, 0/1 비트 변환과 선형 변환입니다. 예를 들어, k를 0으로 설정하면 함수 값이 1이 될 것입니다. 따라서 0/1 비트 변환을 정확하게 복구 할 수 있습니다.

**7:15**

If you set k to very large number on the other hand, it's going to look more like the linear transformation function.

**7:24**

So in this sense, this transformation is very flexible. It allows us to control the shape of the transformation. It also has a nice property of the upper bound.

**7:38**

And this upper bound is useful to control the inference of a particular term.

**7:43**

And so that we can prevent a spammer from just increasing the count of one term to spam all queries that might match this term.

반면 k를 매우 큰 수로 설정하면 선형 변환 함수처럼 보일 것입니다. 이런 점에서이 변환은 매우 유연합니다. 그것은 우리가 변형의 모양을 제어 할 수있게 해줍니다. 그것은 또한 상한의 좋은 속성을 가지고 있습니다. 그리고이 상한은 특정 용어의 추론을 제어하는 데 유용합니다. 따라서 스팸 발송자가이 용어와 일치하는 모든 검색어를 스팸 메일로 보내는 것을 방지 할 수 있습니다.

**7:57**

In other words, this upper bound might also ensure that all terms would be counted when we aggregate the weights to compute the score.

**8:06**

As I said, this transformation function has worked well so far.

즉,이 상한은 가중치를 집계하여 점수를 계산할 때 모든 용어가 계산되도록 할 수 있습니다. 제가 말했듯이,이 변환 함수는 지금까지 잘 작동했습니다.

**8:12**

So to summarize this lecture, the main point is that we need to do Sublinear TF Transformation, and this is needed to capture the intuition of diminishing return from higher term counts.

이 강의를 요약하기 위해, 요점은 우리가 Sublinear TF 변환을해야한다는 것입니다. 이것은 높은 학기 수에서의 수익 감소의 직관을 포착하는 데 필요합니다.

**8:26**

It's also to avoid the dominance by one single term over all others. This BM25 transformation that we talked about is very interesting. It's so far one of the best-performing TF Transformation formulas. It has upper bound, and so it's also robust and effective.

또한 다른 모든 것보다 단일 용어로 지배력을 피하기 위해서입니다. 우리가 얘기 한이 BM25 변형은 매우 흥미 롭습니다. 지금까지 가장 우수한 TF 변환 공식 중 하나입니다. 상한선이 있기 때문에 강력하고 효과적입니다.

**8:47**

Now if we're plugging this function into our TF-IDF weighting vector space model. Then we'd end up having the following ranking function, which has a BM25 TF component.

이제 이 함수를 TF-IDF 가중치 벡터 공간 모델에 연결하면됩니다. 그런 다음 BM25 TF 구성 요소가있는 다음 순위 지정 기능을 갖게됩니다.

**9:01**

Now, this is already very close to a state of the odd ranking function called BM25. And we'll discuss how we can further improve this formula in the next lecture.

이제는 이미 BM25라고 불리는 이상한 순위 함수의 상태에 매우 가깝습니다. 그리고 우리는 다음 강의에서이 수식을 더 향상시킬 수있는 방법을 논의 할 것입니다.

**2.3-TR-Doc\_Length\_Normalization**

**0:08**

This lecture is about Document Length Normalization in the Vector Space Model. In this lecture, we will continue the discussion of the vector space model. In particular, we're going to discuss the issue of document length normalization.

이 강의는 벡터 공간 모델에서 문서 길이 정규화에 관한 것입니다. 이 강의에서 우리는 벡터 공간 모델에 대한 논의를 계속할 것입니다. 특히 문서 길이 정규화 문제에 대해 논의 할 것입니다.

**0:25**

So far in the lectures about the vector space model, we have used the various signals from the document to assess the matching of the document with a query. In particular, we have considered the tone frequency. The count of a tone in a document. We have also considered it's global statistics such as, IDF, Inverse Document Frequency. But we have not considered document lengths.

지금까지 벡터 공간 모델에 대한 강의에서 문서의 다양한 신호를 사용하여 문서와 쿼리의 일치를 평가했습니다. 특히 톤 주파수를 고려했습니다. 문서의 톤 수입니다. 또한 IDF, 역 문서 빈도와 같은 글로벌 통계를 고려했습니다. 하지만 문서 길이는 고려하지 않았습니다.

**0:54**

So here I show two example documents, d4 is much shorter with only 100 words.

**1:01**

D6 on the other hand, has a 5000 words. If you look at the matching of these query words, we see that in d6, there are more matchings of the query words. But one might reason that, d6 may have matched these query words in a scattered manner.

**1:24**

So maybe the topic of d6, is not really about the topic of the query.

**1:31**

So, the discussion of the campaign at the beginning of the document, may have nothing to do with the managing of presidential at the end.

그래서 여기에 두 개의 예제 문서를 보여 드리겠습니다. d4는 100 단어로 훨씬 짧습니다. D6는 5000 단어입니다. 이러한 쿼리 단어의 일치를 보면 d6에서 쿼리 단어가 더 일치한다는 것을 알 수 있습니다. 그러나 d6가 이러한 검색어를 흩어져있는 방식으로 일치 시켰을 수도 있습니다. 아마 d6의 주제는 실제로 쿼리 주제에 관한 것이 아닙니다. 따라서 문서 초반의 캠페인에 대한 논의는 결국 대통령 선거 관리와 관련이 없을 수도 있습니다.

**1:40**

In general, if you think about the long documents, they would have a higher chance for matching any query. In fact, if you generate a long document randomly by assembling words from a distribution of words, then eventually you probably will match an inquiry.

**2:00**

So in this sense, we should penalize on documents because they just naturally have better chance matching to any query, and this is idea of document normalization.

일반적으로 긴 문서를 생각하면 어떤 쿼리와도 일치 할 가능성이 더 높습니다. 사실 단어의 분포에서 단어를 조합하여 긴 문서를 무작위로 생성하면 궁극적으로는 문의와 일치하게됩니다. 따라서 이러한 의미에서 우리는 자연스럽게 모든 쿼리와 일치하는 기회가 더 많기 때문에 문서에 불이익을 주어야합니다. 이는 문서 정규화의 아이디어입니다.

**2:12**

We also need to be careful in avoiding to over penalize long documents.

**2:19**

On the one hand, we want to penalize the long document. But on the other hand, we also don't want to over-penalize them. Now, the reasoning is because a document that may be long because of different reasons.

**2:32**

In one case, the document may be long because it uses more words.

**2:38**

So for example, think about the vortex article on the research paper. It would use more words than the corresponding abstract.

**2:49**

So, this is a case where we probably should penalize the matching of

**2:54**

long documents such as a full paper. When we compare the matching of words in such a long document with matching of the words in the shop abstract.

또한 긴 문서를 과도하게 처벌하지 않도록 조심해야합니다. 한편으로는 긴 문서에 불이익을주고 싶습니다. 그러나 다른 한편으로, 우리는 또한 그들을 과소 평가하고 싶지 않습니다. 이유는 다른 이유 때문에 문서가 오래 걸릴 수 있기 때문입니다. 한 가지 경우에 더 많은 단어를 사용하기 때문에 문서가 길어질 수 있습니다. 예를 들어 연구 논문의 소용돌이에 대해 생각해보십시오. 해당 초록보다 많은 단어를 사용합니다. 따라서 전체 종이와 같은 긴 문서의 매칭을 처벌해야하는 경우입니다. 그런 긴 문서의 단어 매칭을 상점 개요의 단어 매치와 비교할 때.

**3:07**

Then long papers in general, have a higher chance of matching clearer words, therefore, we should penalize them. However, there is another case when the document is long, and that is when the document simply has more content. Now consider another case of long document, where we simply concatenate a lot of abstracts of different papers. In such a case, obviously, we don't want to over-penalize such a long document. Indeed, we probably don't want to penalize such a document because it's long.

그런 다음 일반적으로 긴 논문은 더 명확한 단어를 매치 할 기회가 더 많으므로이를 처벌해야합니다. 그러나 문서가 길면 문서가 더 많은 콘텐츠가있는 경우도 있습니다. 이제 다른 문서의 요약을 많이 연결하는 긴 문서의 또 다른 사례를 생각해보십시오. 그러한 경우에, 분명히, 우리는 그러한 긴 문서를 과도하게 처벌하고 싶지 않습니다. 사실, 우리는 오래 전에 그러한 문서를 벌벌로 치고 싶지 않을 것입니다.

**3:39**

So that's why, we need to be careful about using the right degree of penalization.

그래서 우리는 적절한 정도의 벌칙을 적용 할 때주의해야합니다.

**3:48**

A method of that has been working well, based on recent results, is called a pivoted length normalization. And in this case, the idea is to use the average document length as a pivot, as a reference point. That means we'll assume that for the average length documents, the score is about right so the normalizer would be 1. But if the document is longer than the average document length,

**4:14**

then there will be some penalization. Whereas if it's a shorter, then there is even some reward. So this is illustrated at using this slide, on the axis, x-axis you can see the length of document. On the y-axis, we show the normalizer. In this case, the Pivoted Length Normalization formula for the normalizer, is seeing to be interpolation of 1 and the normalize the document in length controlled by a parameter B here.

최근의 결과를 토대로 피벗 된 길이 정규화라고하는 방법이 잘 작동합니다. 이 경우, 평균 포인트 길이를 기준 포인트로 사용하는 것이 좋습니다. 평균 길이 문서의 경우 스코어가 거의 올바르므로 정규화 도구가 1이라고 가정합니다. 그러나 문서가 평균 문서 길이보다 길면 몇 가지 불이익이 있습니다. 반대로 짧으면 보상도 있습니다. 그래서이 슬라이드를 사용하여 그림이 그려져 있습니다. 축에서 x 축으로 문서의 길이를 볼 수 있습니다. y 축에서 노멀 라이저를 보여줍니다. 이 경우 노멀 라이저의 피벗 길이 정규화 수식은 1의 보간으로 간주되며 여기에서 매개 변수 B로 제어되는 길이로 문서를 정규화합니다.

**4:53**

So you can see here, when we first divide the length of the document by the average documents, this not only gives us some sense about how this document is compared with average documents, but also gives us a benefit of not worrying about the unit of length. We can measure the length by words or by characters.

여기에서 볼 수 있듯이 문서의 길이를 평균 문서로 나누면이 문서를 평균 문서와 비교할 때 어떤 의미가 있는지뿐만 아니라이 문서의 평균 단위를 걱정하지 않아도됩니다. 길이. 단어 나 문자로 길이를 측정 할 수 있습니다.

**5:20**

Anyway, this normalizer has interesting property. First we see that, if we set the parameter b to 0 then the value would be 1. So, there's no lens normalization at all. So, b, in this sense, controls the lens normalization.

어쨌든,이 normalizer는 흥미로운 속성을 가지고 있습니다. 먼저 매개 변수 b를 0으로 설정하면 값이 1이됩니다. 따라서 렌즈 정규화가 전혀 없습니다. 따라서 b는 이러한 의미에서 렌즈 정규화를 제어합니다.

**5:39**

Whereas, if we set b to a nonzero value, then the normalizer would look like this. All right, so the value would be higher for documents that are longer than the average document lens.

반대로 b를 0이 아닌 값으로 설정하면 노멀 라이저는 다음과 같이 보입니다. 좋습니다. 따라서 평균 문서 렌즈보다 긴 문서의 경우 값이 더 높아집니다.

**5:53**

Whereas, the value of the normalizer would be shorter, would be smaller for shorter documents. So in this sense, we see there is a penalization for long documents, and there's a reward for short documents.

반면, 노멀 라이저의 값은 더 짧을 것이고, 더 짧은 문서의 경우 더 작을 것입니다. 그래서이 의미에서 우리는 긴 문서에 대한 벌칙이 있고, 짧은 문서에 대한 보상이 있음을 알 수 있습니다.

**6:09**

The degree of penalization is controlled by b, because if we set b to a larger value, then the normalizer would look like this. There's even more penalization for long documents and more reward for the short documents. By adjusting b, which varies from 0 to 1, we can control the degree of length normalization. So, if we plug in this length normalization fact that into the vector space model, ranking functions is that we have already examined them.

벌칙의 정도는 b에 의해 제어됩니다. 왜냐하면 b를 더 큰 값으로 설정하면 노멀 라이저는 다음과 같이 보일 것이기 때문입니다. 긴 문서에는 벌점이 많고 짧은 문서에는 더 많은 보상이 있습니다. 0에서 1까지 변화하는 b를 조정함으로써 길이 정규화의 정도를 제어 할 수 있습니다. 따라서 벡터 공간 모델에이 길이 정규화 사실을 연결하면 순위 함수는 이미 살펴 보았습니다.

**6:41**

Then we will end up having the following formulas.

**6:46**

And these are in fact the state of the vector space model formulas. Let's take a look at each of them. The first one is called a pivoted length normalization vector space model, and a reference in [INAUDIBLE] duration of this model. And here we see that, it's basically a TFI model that we have discussed, the idea of component should be very familiar to you.

그러면 다음과 같은 공식을 갖게 될 것입니다. 그리고 이것들은 사실 벡터 공간 모델 공식의 상태입니다. 각각을 살펴 보겠습니다. 첫 번째 것은 피벗 된 길이 정규화 벡터 공간 모델이고이 모델의 [INAUDIBLE] 지속 시간에 대한 참조입니다. 그리고 여기서 우리는 기본적으로 우리가 논의한 TFI 모델이라는 것을 알았습니다. 구성 요소에 대한 아이디어는 당신에게 매우 친숙해야합니다.

**7:18**

There is also a query term frequency component here.

**7:24**

And then, in the middle, there is the normalizer tf and in this case, we see we use the double logarithm as we discussed before and this is to achieve a sublinear transformation. But we also put a document the length normalizer in the bottom. Right, so this would cause penalization for long document, because the larger the denominator is, then the smaller the is. And this is of course controlled by the parameter b here.

**8:01**

And you can see again, if b is set to 0 then there is no length normalization.

또한 쿼리 용어 빈도 구성 요소가 여기에 있습니다. 그리고 중간에 표준화 기 tf가 있습니다.이 경우 앞에서 설명한 것처럼 double logarithm을 사용하는 것을 볼 수 있습니다. 이것은 서브 선형 변환을 수행하는 것입니다. 그러나 우리는 또한 길이 normalizer를 문서 하단에 넣습니다. 그렇기 때문에 긴 문서의 경우 불이익을 초래할 수 있습니다. 왜냐하면 분모가 커질수록 크기가 작아지기 때문입니다. 그리고 이것은 물론 파라미터 b에 의해 제어됩니다. 그리고 다시 볼 수 있습니다. b가 0으로 설정된 경우 길이 정규화가 없습니다.

**8:08**

Okay, so this is one of the two most effective at these base model formulas. The next one called a BM25 or Okapi, is also similar in that it also has a IDF component here, and query IDF component here.

**8:32**

But in the middle, the normal issue's a little bit different. As we explained, there is our copy tf transformation here, and that does sublinear transformation with the upper bound.

좋아요, 그래서 이것은이 기본 모델 수식에서 가장 효과적인 두 가지 중 하나입니다. BM25 또는 Okapi라고 부르는 다음 번에는 IDF 구성 요소가 있고 IDF 구성 요소를 여기에서 쿼리한다는 점에서 비슷합니다. 그러나 중간에, 정상적인 문제는 조금 다릅니다. 우리가 설명했듯이 여기에 우리의 복사 tf 변환이 있으며, 그것은 상한선을 가진 sublinear 변환을합니다.

**8:48**

In this case we have put the length normalization factor here. We're adjusting k but it achieves a similar factor, because we put a normalizer in the denominator. Therefore, again, if a document is longer then the term weight will be smaller.

**9:10**

So you can see after we have gone through all the n answers that we talked about, and we have in the end reached the basically the state of god functions. So, So far, we have talked about mainly how to place the document vector in the vector space.

이 경우 길이 정규화 계수를 여기에 입력했습니다. 우리는 k를 조정하고 있지만 비슷한 요소를 얻습니다. 왜냐하면 정규화기를 분모에 넣었 기 때문입니다. 따라서 문서가 더 길면 용어의 가중치가 작아집니다. 그래서 우리는 우리가 말한 모든 대답을 모두 마친 후에 볼 수 있습니다. 결국 우리는 기본적으로 하나님의 기능의 상태에 도달했습니다. 그래서 지금까지 우리는 주로 문서 벡터를 벡터 공간에 배치하는 방법에 대해 이야기했습니다.

**9:35**

And, this has played an important role in determining the effectiveness of the simple function. But there are also other dimensions, where we did not really examine details. For example, can we further improve the instantiation of the dimension of the Vector Space Model? Now, we've just assumed that the bag of words representation should issue dimension as a word but obviously, we can see there are many other choices. For example, a stemmed word, those are the words that haven't transformed into the same root form, so that computation and computing were all become the same and they can be match. We get those stop word removal. This is to remove some very common words that don't carry any content like the off.

그리고 이것은 단순한 기능의 효과를 결정하는 데 중요한 역할을했습니다. 그러나 세부 사항을 실제로 검토하지 않은 다른 차원도 있습니다. 예를 들어 벡터 공간 모델의 차원 인스턴스화를 더 향상시킬 수 있습니까? 지금, 우리는 방금 단어 표현의 가방이 단어로 차원을 발행해야한다고 생각했지만 분명히 많은 다른 선택 사항이 있음을 알 수 있습니다. 예를 들어 줄기가있는 단어는 동일한 루트 형식으로 변환되지 않은 단어이므로 계산과 계산이 모두 동일 해지고 일치 할 수 있습니다. 우리는 그 말장난 제거를 얻는다. 이것은 오프와 같은 컨텐트를 포함하지 않는 매우 일반적인 단어를 제거하는 것입니다.

**10:26**

We get use of phrases to define dimensions. We can even use later in the semantical analysis, it will find some clusters of words that represent the a late in the concept as one by an engine.

**10:39**

We can also use smaller unit, like a character end grams those are sequences of and the characters for dimensions.

**10:50**

However, in practice, people have found that the bag-of-words representation with phrases is still the most effective one and it's also efficient. So, this is still so far the most popular dimension instantiation method.

**11:10**

And it's used in all major search engines.

우리는 구를 사용하여 차원을 정의합니다. 의미 론적 분석에서 나중에 사용할 수도 있습니다. 개념의 후반 부분을 엔진으로 나타내는 단어 군집을 찾습니다. 우리는 또한 작은 단위를 사용할 수 있습니다. 예를 들어 문자 끝 그램은 시퀀스이고 문자는 차원입니다. 그러나 실제로, 사람들은 어구 표현을 사용하는 백 단어 표현이 여전히 가장 효과적인 것으로 밝혀졌으며 또한 효율적입니다. 따라서 이것은 아직까지는 가장 대중적인 차원 인스턴스화 방법입니다. 그리고 그것은 모든 주요 검색 엔진에 사용됩니다.

**11:13**

I should also mention, that sometimes we need to do language specific and domain specific tokenization. And this is actually very important, as we might have variations of terms that might prevent us from matching them with each other, even when they mean the same thing. In some languages like Chinese, there is also the challenge in segmenting

**11:40**

text to obtain word band rates because it's just a sequence of characters. A word might correspond to one character or two characters or even three characters. So, it's easier in English when we have a space to separate the words. In some other languages, we may need to do some Americanize processing to figure a way out of what are the boundaries for words. There is also the possibility to improve the similarity of the function. And so far we have used as a top product, but one can imagine there are other measures. For example, we can measure the cosine of the angle between two vectors. Or we can use Euclidean distance measure.

언어 별 및 도메인 별 토큰 화를 수행해야하는 경우도 있음을 언급해야합니다. 실제로 이것은 동일한 의미를 지니고 있어도 서로 일치시키는 것을 막을 수있는 용어의 변형이있을 수 있기 때문에 실제로 매우 중요합니다. 중국어와 같은 일부 언어에서는 문자 시퀀스이므로 단어 대역 비율을 얻기 위해 텍스트를 분할하는 데 어려움이 있습니다. 단어는 한 문자 또는 두 문자 또는 세 문자와 일치 할 수 있습니다. 따라서 단어를 분리 할 공간이있을 때 영어가 쉬워집니다. 다른 언어에서는 단어의 경계를 벗어나는 방법을 찾아 내기 위해 일부 Americanize 처리가 필요할 수 있습니다. 함수의 유사성을 향상시킬 수도 있습니다. 그리고 지금까지 우리는 최고 제품으로 사용했지만 다른 방법이 있다고 상상할 수 있습니다. 예를 들어 두 벡터 사이의 각의 코사인을 측정 할 수 있습니다. 또는 유클리드 거리 측정을 사용할 수 있습니다.

**12:24**

And these are all possible, but dot product seems still the best and one reason is because it's very general.

**12:33**

In fact that it's sufficiently general, if you consider the possibilities of doing waiting in different ways.

**12:44**

So, for example, cosine measure can be thought of as the thought product of two normalized factors. That means, we first normalize each factor and then we take the thought product. That would be critical to the cosine measure. I just mentioned that the BM25, seems to be one of the most effective formulas.

그리고 이것들은 모두 가능하지만, 점 제품은 여전히 최고로 보입니다. 그리고 한 가지 이유는 매우 일반적이기 때문입니다. 사실 그것은 충분히 일반적입니다. 다른 방법으로 기다리는 것이 가능한지 생각해보십시오. 예를 들어, 코사인 측정은 두 개의 정규화 된 요소의 사고 산출물로 생각할 수 있습니다. 즉, 우리는 먼저 각 요소를 표준화 한 다음 생각 제품을 사용합니다. 그것은 코사인 측정에 중요합니다. BM25가 가장 효과적인 공식 중 하나 인 것 같습니다.

**13:04**

But there has been also further developments in improving BM25. Although, none of these words have changed the BM25 fundamental. So in one line work, people have divide the BM25 F. Here, F stands for field, and this is use BM25 for documents with structures. So for example, you might consider a title field, the abstract, or body of the research article. Or even anchor text on the web page, those are the text fields that describe links to other pages and these can all be combined with a proper way of different fields to help improve scoring for different documents. When we use BM25 for such a document and the obvious choice is to apply BM25 for each field and then combine the scores. Basically, the idea of BM25F is to first combine the frequency counts of terms in all the fields, and then apply BM25. Now, this has advantage of avoiding over counting the first occurrence of the term. Remember in the sublinear transformation of TF, the first occurrence is very important and it contributes a large weight. And if we do that for all the fields, then the same term might have gained a lot of advantage in every field. But when we combine these word frequencies together, we just do the transformation one time. At that time, then the extra occurrences will not be counted as fresh first recurrences.

그러나 BM25의 개선에 대한 발전이있었습니다. 그러나,이 단어들 중 어느 것도 BM25 기본을 바 꾸었습니다. 따라서 한 줄의 작업에서 사람들은 BM25 F를 나눕니다. 여기서 F는 필드를 나타내며, 구조가있는 문서의 경우 BM25를 사용합니다. 예를 들어 제목 필드, 요약 또는 연구 기사의 본문을 고려할 수 있습니다. 또는 웹 페이지에 텍스트를 앵커 할 수도 있습니다.이 텍스트 필드는 다른 페이지에 대한 링크를 설명하는 텍스트 필드이며 서로 다른 필드의 적절한 방법으로 결합하여 서로 다른 문서의 점수를 높일 수 있습니다. BM25를 그러한 문서에 사용할 때 명백한 선택은 각 필드에 대해 BM25를 적용한 다음 점수를 결합하는 것입니다. 기본적으로 BM25F의 아이디어는 모든 분야의 용어 빈도 카운트를 먼저 결합한 다음 BM25를 적용하는 것입니다. 자, 이것은 용어의 첫 번째 발생을 계산하는 것을 피하는 이점이 있습니다. TF의 sublinear 변형에서 기억하십시오, 첫번째 발생은 아주 중요하고 큰 무게를 공헌합니다. 우리가 모든 분야에 대해 그렇게한다면, 같은 용어는 모든 분야에서 많은 이점을 얻었을 것입니다. 그러나 우리가이 단어의 빈도를 결합 할 때, 우리는 한 번 변환을합니다. 그 때 여분의 사건은 새로운 첫 번째 재발로 간주되지 않습니다.

**14:48**

And this method has been working very well for scoring structure with documents.

**14:55**

The other line of extension is called a BM25+. In this line, risk is to have to address the problem of over penalization of long documents by BM25.

**15:08**

So to address this problem, the fix is actually quite simple. We can simply add a small constant to the TF normalization formula. But what's interesting is that, we can analytically prove that by doing such a small modification, we will fix the problem of over penalization of law documents by the original BM25. So the new formula called BM25+, is empirically and analytically shown to be better than BM25.

그리고이 방법은 문서로 구조를 채점하는데 매우 효과적입니다. 다른 연장선은 BM25 +라고합니다. 이 라인에서 BM25는 긴 문서의 과다 처벌 문제를 해결해야 할 위험이 있습니다. 따라서이 문제를 해결하기 위해 실제로는 수정이 간단합니다. 우리는 간단히 TF 정규화 공식에 작은 상수를 추가 할 수 있습니다. 그러나 흥미로운 점은 분석을 통해 이러한 작은 수정 작업을 통해 원래 BM25의 법률 문서 처벌 초과 문제를 해결할 수 있다는 것입니다. 그래서 BM25 +라고 불리는 새로운 공식은 경험적으로 분석적으로 BM25보다 더 나은 것으로 나타났습니다.

**15:42**

So to summarize all what we have said about vector space model, here are the major take away points. First, in such a model, we use the similarity of relevance. Assuming that relevance of a document with respect to a query, is

**16:02**

basically proportional to the similarity between the query and the document. So naturally, that implies that the query and document must have been represented in the same way. And in this case, we will present them as vectors in high-dimensional vector space. Where the dimensions are defined by words, or concepts, or terms, in general.

**16:25**

And we generally, need to use a lot of heuristics to design the ranking function. We use some examples, which show the needs for several heuristics, including Tf weighting and transformation.

그래서 우리가 벡터 공간 모델에 대해 말한 것을 요약하기 위해, 여기에는 주요 제거 점이 있습니다. 첫째, 이러한 모델에서 우리는 관련성의 유사성을 사용합니다. 쿼리와 관련된 문서의 관련성이 기본적으로 쿼리와 문서 간의 유사성에 비례한다고 가정합니다. 당연히 이는 쿼리와 문서가 같은 방식으로 표현되어 있어야 함을 의미합니다. 이 경우, 우리는 그것들을 고차원 벡터 공간의 벡터로 제시 할 것입니다. 차원이 단어, 개념 또는 용어로 정의되는 곳. 그리고 우리는 일반적으로 순위 결정 기능을 설계하기 위해 많은 경험적 방법을 사용해야합니다. 우리는 몇 가지 예를 사용합니다. Tf 가중치 지정과 변형을 포함한 여러 가지 경험적 방법의 필요성을 보여줍니다.

**16:38**

And IDF weighting, and document length normalization. These major heuristics are the most important of heuristics, to ensure such a general ranking function to work well for all kinds of test. And finally, BM25 and pivoted normalization seem to be the most effective formulas out of the vector space model. Now I have to say that, I put BM25 in the category of vector space model, but in fact, the BM25 has been derived using probabilistic model.

IDF 가중치 및 문서 길이 정규화. 이러한 주요 발견 적 방법은 모든 종류의 테스트에서 잘 작동하는 일반적인 순위 지정 기능을 보장하기 위해 가장 중요한 휴리스틱 스입니다. 마지막으로 BM25와 피벗 표준화가 벡터 공간 모델에서 가장 효과적인 공식 인 것으로 보입니다. 이제 저는 BM25를 벡터 공간 모델 범주에 넣었지만 사실 BM25는 확률 론적 모델을 사용하여 파생되었습니다.

**17:11**

So the reason why I've put it in the vector space model is first, the ranking function actually has a nice interpretation in the vector space model. We can easily see, it looks very much like a vector space model, with a special waiting function.

그래서 이것을 벡터 공간 모델에 넣은 이유는 첫째로, 순위 함수는 실제로 벡터 공간 모델에서 좋은 해석을합니다. 우리는 쉽게 볼 수 있습니다. 특별한 대기 함수를 가진 벡터 공간 모델과 매우 흡사합니다.

**17:28**

The second reason is because the original BM25, has somewhat different form of IDF.

두 번째 이유는 원래의 BM25가 다소 다른 형태의 IDF를 가지고 있기 때문입니다.

**17:36**

And that form of IDF after the [INAUDIBLE] doesn't work so well as the standard IDF that you have seen here. So as effective retrieval function, BM25 should probably use a heuristic modification of the IDF. To make them even more look like a vector space model

**17:59**

There are some additional readings. The first is, a paper about the pivoted length normalization. It's an excellent example of using empirical data analysis to suggest the need for length normalization and then further derive the length normalization formula. The second, is the original paper where the BM25 was proposed.

**18:24**

The third paper, has a thorough discussion of BM25 and its extensions, particularly BM25 F.

**18:32**

And finally, in the last paper has a discussion of improving BM25 to correct the over penalization of long documents.

[IDF] 이후의 IDF 양식은 여기에서 본 표준 IDF만큼 잘 작동하지 않습니다. 효과적인 검색 기능으로, BM25는 아마도 IDF의 휴리스틱 수정을 사용해야합니다. 벡터 공간 모델처럼 보이게 만듭니다. 몇 가지 추가 판독 값이 있습니다. 첫 번째는 피벗 된 길이 정규화에 대한 종이입니다. 이것은 경험적 데이터 분석을 사용하여 길이 정규화의 필요성을 제안하고 길이 정규화 공식을 더 유도하는 훌륭한 예입니다. 두 번째는 BM25가 제안 된 원본 종이입니다. 세 번째 논문은 BM25와 그 확장, 특히 BM25 F에 대한 철저한 논의가 있습니다. 마지막으로 마지막 문서에서 긴 문서의 과다한 처벌을 수정하기위한 BM25 개선에 대한 논의가 있습니다.

**2.4-Implementation-of-TR-Systems**

**0:12**

In this lecture we will discuss how we can implement a text retrieval method to build a search engine. The main challenge is to manage a lot of text data and to enable a query to be answered very quickly and to respond to many queries. This is a typical text retrieval system architecture. We can see the documents are first processed by a tokenizer to get tokenized units, for example, words. And then, these words, or tokens, will be processed by a indexer that will create a index, which is a data structure for the search engine to use to quickly answer a query. And the query would be going through a similar processing step. So the Tokenizer would be apprised of the query as well, so that the text can be processed in the same way. The same units would be matched with each other. The query's representation would then be given to the Scorer, which would use the index to quickly answer user's query by scoring the documents and then ranking them. The results will be given to the user. And then the user can look at the results and provided us some feedback that can be explicit judgements of both which documents are good, which documents are bad. Or implicit feedback such as so that user didn't have to do anything extra. End user will just look at the results, and skip some, and click on some result to view. So these interacting signals can be used by the system to improve the ranking accuracy by assuming that viewed documents are better than the skipped ones. So a search engine system then can be divided into three parts. The first part is the indexer, and the second part is a Scorer that responds to the users query, and the third part is a Feedback mechanism. Now typically, the Indexer is done in the offline manner, so you can pre-process the correct data and to build the inventory index, which we will introduce in moment. And this data structure can then be used by the online module which is a scorer to process a user's query dynamically and quickly generate search results. The feedback mechanism can be done online or offline, depending on the method. The implementation of the indexer and the scorer is very standard, and this is the main topic of this lecture and the next few lectures. The feedback mechanism, on the other hand, has variations, it depends on which method is used. So that is usually done in algorithms specific way. Let's first talk about the tokenizer. Tokernization is a normalized lexical units in through the same form, so that semantically similar words can be matched with each other. Now, in the language like English, stemming is often used and this will map all the inflectional forms of words into the same root form. So for example, computer, computation, and computing can all be matched to the root form compute. This way all these different forms of computing can be matched with each other.

이 강의에서는 검색 엔진을 구축하기 위해 텍스트 검색 방법을 구현하는 방법에 대해 설명합니다. 주요 과제는 많은 텍스트 데이터를 관리하고 쿼리가 매우 신속하게 응답하고 많은 쿼리에 응답 할 수있게하는 것입니다. 이것은 일반적인 텍스트 검색 시스템 아키텍처입니다. 문서는 토크 나이저로 먼저 처리되어 토큰 화 된 단위 (예 : 단어)를 얻을 수 있습니다. 그런 다음 이러한 단어 나 토큰은 색인을 생성하는 인덱서에 의해 처리됩니다.이 색인은 검색 엔진이 쿼리에 신속하게 응답하는 데 사용하는 데이터 구조입니다. 쿼리는 비슷한 처리 단계를 거칩니다. 그래서 Tokenizer는 질의에 대해서도 알게 될 것이므로 텍스트는 같은 방식으로 처리 될 수 있습니다. 동일한 단위가 서로 일치합니다. 쿼리의 표현은 Scorer에게 주어지며 Scorer는 인덱스를 사용하여 문서를 채점하고 순위를 지정하여 사용자 쿼리에 신속하게 응답합니다. 결과는 사용자에게 제공됩니다. 그런 다음 사용자는 결과를보고 어떤 문서가 좋고 어떤 문서가 나쁜지에 대한 명시적인 판단이 될 수있는 피드백을 제공 할 수 있습니다. 또는 사용자가 추가 작업을 수행 할 필요가 없도록 암시적인 피드백. 최종 사용자는 결과를보고 건너 뛰고 일부 결과를 클릭하면 볼 수 있습니다. 따라서 이러한 상호 작용하는 신호는 조회 된 문서가 건너 뛴 문서보다 낫다고 가정하여 순위 정확도를 향상시키기 위해 시스템에서 사용할 수 있습니다. 그래서 검색 엔진 시스템은 세 부분으로 나눌 수 있습니다. 첫 번째 부분은 인덱서이고 두 번째 부분은 사용자 쿼리에 응답하는 Scorer이고 세 번째 부분은 피드백 메커니즘입니다. 이제는 일반적으로 인덱서가 오프라인 방식으로 수행되므로 올바른 데이터를 사전 처리하고 인벤토리 색인을 작성하는 것이 가능합니다. 그리고이 데이터 구조는 사용자의 쿼리를 동적으로 처리하고 검색 결과를 신속하게 생성하기 위해 스코어러 인 온라인 모듈에 의해 사용될 수 있습니다. 피드백 메커니즘은 방법에 따라 온라인 또는 오프라인으로 수행 할 수 있습니다. 인덱서와 득점 자의 구현은 매우 표준 적이며이 강의의 주요 주제이자 다음 강의입니다. 반대로 피드백 메커니즘에는 변형이 있으며 어떤 방법이 사용되는지에 따라 다릅니다. 그래서 보통 알고리즘의 특정 방식으로 수행됩니다. 먼저 토크 나이저에 대해 이야기 해 봅시다. 토큰 화는 동일한 형식을 통해 정규화 된 어휘 단위이므로 의미 상으로 유사한 단어를 서로 일치시킬 수 있습니다. 이제는 영어와 같은 언어에서 형태소 분석이 종종 사용되며 이는 모든 굴절 형 단어를 동일한 루트 형태로 매핑합니다. 예를 들어, 컴퓨터, 계산 및 컴퓨팅은 모두 루트 양식 계산과 일치 할 수 있습니다. 이런 식으로 모든 다른 형태의 컴퓨팅은 서로 일치 될 수 있습니다.

Now normally, this is a good idea, to increase the coverage of documents that are matched up with this query. But it's also not always beneficial, because sometimes the subtlest difference between computer and computation might still suggest the difference in the coverage of the content. But in most cases, stemming seems to be beneficial. When we tokenize the text in some other languages, for example Chinese, we might face some special challenges in segmenting the text to find the word boundaries. Because it's not obvious where the boundary is as there's no space to separate them. So here of course, we have to use some language specific processing techniques. Once we do tokenization, then we would index the text documents and than it'll convert the documents and do some data structure that can enable faster search. The basic idea is to precompute as much as we can basically. So the most commonly used index is call an Inverted index. And this has been used in many search engines to support basic search algorithms. Sometimes the other indices, for example, document index might be needed in order to support feedback, like I said. And these kind of techniques are not really standard in that they vary a lot according to the feedback methods. To understand why we want to use inverted index it will be useful for you to think about how you would respond to a single term query quickly. So if you want to use more time to think about that, pause the video. So think about how you can pre process the text data so that you can quickly respond to a query with just one word. Where if you have thought about that question, you might realize that where the best is to simply create the list of documents that match every term in the vocabulary. In this way, you can basically pre-construct the answers. So when you see a term you can simply just to fetch the random list of documents for that term and return the list to the user. So that's the fastest way to respond to a single term here.

이제는 일반적으로이 쿼리와 일치하는 문서의 적용 범위를 늘리는 것이 좋습니다. 그러나 컴퓨터와 계산 사이의 가장 미묘한 차이로 인해 여전히 콘텐츠의 범위가 달라질 수 있기 때문에 언제나 유익한 것은 아닙니다. 그러나 대부분의 경우, 형태소 분석이 유익한 것으로 보입니다. 텍스트를 다른 언어 (예 : 중국어)로 토큰화할 때 단어 경계를 찾기 위해 텍스트를 분할 할 때 몇 가지 특별한 문제가 발생할 수 있습니다. 경계를 구분할 여지가 없으므로 경계가 명확하지 않기 때문입니다. 따라서 여기에서는 물론 언어 별 처리 기술을 사용해야합니다. 일단 토큰 화를하면 텍스트 문서를 색인화하고 문서를 변환하고 더 빠른 검색을 가능하게하는 일부 데이터 구조를 작성합니다. 기본 아이디어는 기본적으로 미리 계산하는 것입니다. 따라서 가장 일반적으로 사용되는 인덱스는 Inverted 인덱스를 호출하는 것입니다. 그리고 이것은 많은 검색 엔진에서 기본 검색 알고리즘을 지원하는 데 사용되었습니다. 내가 말했던 것처럼 피드백을 지원하기 위해 다른 색인, 예를 들어 문서 색인이 필요할 때가 있습니다. 그리고 이런 종류의 기법은 피드백 방식에 따라 많이 달라 지므로 실제로 표준이 아닙니다. 반전 된 인덱스를 사용하려는 이유를 이해하려면 단일 용어 쿼리에 신속하게 대응하는 방법에 대해 생각하는 것이 유용 할 것입니다. 따라서 생각할 시간을 더 많이 사용하려면 비디오를 일시 중지하십시오. 따라서 텍스트 데이터를 사전 처리하여 한 단어로 신속하게 쿼리에 응답 할 수있는 방법에 대해 생각해보십시오. 그 질문에 대해 생각해 본다면, 어휘의 모든 용어와 일치하는 문서 목록을 만드는 것이 가장 좋은 경우를 알 수 있습니다. 이런 방식으로, 기본적으로 답을 미리 구성 할 수 있습니다. 따라서 용어를 볼 때 단순히 해당 용어에 대한 문서의 무작위 목록을 가져 와서 목록을 사용자에게 반환 할 수 있습니다. 이것이 여기서 단 하나의 용어에 가장 빨리 응답하는 방법입니다.

Now the idea of the invert index is actually, basically, like that. We're going to do pre-constructed search an index, that will allows us to quickly find all the documents that match a particular term. So let's take a look at this example. We have three documents here, and these are the documents that you have seen in some previous lectures. Suppose that we want to create an inverted index for these documents. Then we want to maintain a dictionary, in the dictionary we will have one entry for each term and we're going to store some basic statistics about the term. For example, the number of documents that match the term, or the total number of code or frequency of the term, which means we would kind of duplicate the occurrences of the term. And so, for example, news, this term occur in all the three documents, so the count of documents is three. And you might also realize we needed this count of documents, or document frequency, for computing some statistics to be used in the vector space model. Can you think of that? So what weighting heuristic would need this count. Well, that's the idea, right, inverse document frequency. So, IDF is the property of a term, and we can compute it right here. So, with the document that count here, it's easy to compute the idea of, either at this time, or with the old index, or. At random time when we see a query. Now in addition to these basic statistics, we'll also store all the documents that matched the news, and these entries are stored in the file called Postings.

이제 반전 지수의 개념은 실제로 기본적으로 그렇게됩니다. 사전 구축 된 검색을 수행하여 특정 용어와 일치하는 모든 문서를 빠르게 찾을 수 있습니다. 이제이 예제를 살펴 보겠습니다. 여기에 세 개의 문서가 있으며, 이전 강의에서 본 문서입니다. 이 문서들에 대해 역 색인을 만들고 싶다고 가정 해보십시오. 그런 다음 사전을 유지하고 싶습니다. 사전에 각 용어에 대해 하나의 항목이 있으며 용어에 대한 몇 가지 기본 통계를 저장하게됩니다. 예를 들어, 용어와 일치하는 문서의 수 또는 용어의 전체 코드 또는 빈도. 이는 우리가 용어의 출현을 복제하는 것을 의미합니다. 예를 들어 뉴스라는 용어는 세 개의 모든 문서에서 발생하므로 문서 수가 세 개인 경우입니다. 또한 벡터 공간 모델에서 사용되는 일부 통계를 계산하기 위해 문서 수 또는 문서 빈도가 필요하다는 것을 알 수 있습니다. 그 생각을 할 수 있니? 따라서 어떤 가중치 추론에이 계산이 필요한가? 글쎄, 그게 아이디어 야, 맞아, 역 문서 빈도. 따라서 IDF는 용어의 속성이며 여기서 바로 계산할 수 있습니다. 따라서 여기에 포함 된 문서를 사용하면이 시점에서 또는 이전 색인으로 아이디어를 계산하는 것이 쉽습니다. 임의의 시간에 쿼리를 볼 때. 이제 이러한 기본 통계 외에도 뉴스와 일치하는 모든 문서를 저장하고 이러한 항목은 Postings라는 파일에 저장됩니다.

**8:24**

So in this case it matched three documents and we store information about these three documents here. This is the document id, document 1 and the frequency is 1. The tf is one for news, in the second document it's also 1, et cetera. So from this list, we can get all the documents that match the term news and we can also know the frequency of news in these documents. So, if the query has just one word, news, and we have easily look up to this table to find the entry and go quicker into the postings to fetch all the documents that matching yours. So, let's take a look at another term.

따라서이 경우에는 3 개의 문서와 일치하고 여기에이 세 가지 문서에 대한 정보를 저장합니다. 이것은 문서 ID, 문서 1이며 빈도는 1입니다. tf는 뉴스 용이고 두 번째 문서에서는 1 등입니다. 따라서이 목록에서 뉴스라는 용어와 일치하는 모든 문서를 얻을 수 있으며이 문서의 뉴스 빈도를 알 수 있습니다. 따라서 검색어에 단어가 하나 뿐이고 뉴스가있는 경우이 표를 찾아 항목을 찾고 게시글에 더 빨리 들어가서 귀하의 것과 일치하는 모든 문서를 가져옵니다. 이제 다른 용어를 살펴 보겠습니다.

**9:09**

This time, let's take a look at the word presidential.

이번에는 대통령이라는 단어를 살펴 보겠습니다.

**9:14**

This would occur in only one document, document 3. So the document frequency is 1 but it occurred twice in this document. So the frequency count is two, and the frequency count is used for some other reachable method where we might use the frequency to

**9:34**

assess the popularity of a term in the collection. Similarly we'll have a pointer to the postings here, and in this case, there is only one entry here because

**9:48**

the term occurred in just one document and that's here. The document id is 3 and it occurred twice.

이것은 하나의 문서, 문서 3에서만 발생합니다. 따라서 문서 빈도는 1이지만이 문서에서 두 번 발생했습니다. 따라서 빈도 카운트는 2이고 빈도 카운트는 빈도를 사용하여 콜렉션에서 용어의 인기를 평가할 수있는 다른 도달 가능한 메소드에 사용됩니다. 마찬가지로 여기에 게시물에 대한 포인터가 표시됩니다.이 경우 하나의 문서에서만 용어가 사용되었으므로 여기에 항목이 하나뿐입니다. 문서 ID는 3이며 두 번 발생했습니다.

**9:59**

So this is the basic idea of inverted index. It's actually pretty simple, right?

이것은 역 색인의 기본 개념입니다. 사실 꽤 간단합니다.

**10:06**

With this structure we can easily fetch all the documents that match a term. And this will be the basis for scoring documents for a query. Now sometimes we also want to store the positions of these terms.

이 구조를 사용하면 용어와 일치하는 모든 문서를 쉽게 가져올 수 있습니다. 그리고 이것은 쿼리에 대한 문서 채점의 기초가 될 것입니다. 이제 때때로 우리는이 용어의 위치를 저장하기를 원합니다.

**10:25**

So in many of these cases the term occurred just once in the document. So there's only one position for example in this case.

**10:35**

But in this case, the term occurred twice so there's two positions. Now the position information is very useful for the checking whether the matching of query terms is actually within a small window of, let's say, five words or ten words.

**10:52**

Or, whether the matching of the two query terms is, in fact, a phrase of two words. That this can all be checked quickly by using the position from each.

그래서 이러한 많은 경우에서이 용어는 문서에서 한 번만 발생했습니다. 이 경우 예를 들어 단 하나의 위치 만 있습니다. 그러나이 경우 용어는 두 번 발생하므로 두 가지 입장이 있습니다. 이제 위치 정보는 검색어의 일치가 실제로 5 단어 또는 10 단어의 작은 창 내에 있는지 확인하는 데 매우 유용합니다. 또는 두 개의 검색어의 일치 여부는 사실 두 단어로 이루어집니다. 이것은 각각의 위치를 사용하여 신속하게 확인할 수 있습니다.

**11:05**

So, why is inverted index good for fast search? Well, we just talked about the possibility of using the two answer single-term query. And that's very easy. What about the multiple term queries? Well let's first look at the some special cases of the Boolean query. A Boolean query is basically a Boolean expression like this. So I want the value in the document to match both term A and term B. So that's one conjunctive query. Or I want the web documents to match term A or term B. That's a disjunctive query. But how can we answer such a query by using inverted index?

그렇다면 왜 거꾸로 된 색인은 빠른 검색에 좋은가? 글쎄, 우리는 방금 두 개의 답변을 단일 용어 쿼리를 사용하는 가능성에 대해 이야기. 그리고 그것은 매우 쉽습니다. 여러 검색어에 대해서는 어떻게됩니까? 먼저 부울 쿼리의 특수한 경우를 먼저 살펴 보겠습니다. 부울 쿼리는 기본적으로 이와 같은 부울 식입니다. 따라서 문서의 값을 용어 A와 용어 B 모두와 일치 시키길 원합니다. 이것은 하나의 결합 쿼리입니다. 또는 웹 문서가 용어 A 또는 용어 B와 일치하도록하고 싶습니다. 이것은 분리형 쿼리입니다. 그러나 역 색인을 사용하여 그러한 쿼리에 어떻게 대답 할 수 있습니까?

**11:52**

Well if you think a bit about it, it would be obvious because we have simply fetch all the documents that match term A and also fetch all the documents that match term B. And then just take the intersection to answer a query like A and B. Or to take the union to answer the query A or B. So this is all very easy to answer. It's going to be very quick. Now what about the multi-term keyword query? We talked about the vector space model for example and we will do a match such query with document and generate the score. And the score is based on aggregated term weights. So in this case it's not the Boolean query but the scoring can be actually done in similar way. Basically it's similar to disjunctive Boolean query. Basically, it's like A or B. We take the union of all the documents that match at least one query term and then we would aggregate the term weights. So this is a basic idea of using inverted index for scoring documents in general. And we're going to talk about this in more detail later. But for now, let's just look at the question why is in both index, a good idea? Basically why is more efficient than sequentially just scanning documents. This is the obvious approach. You can just compute a score for each document and then you can then sort them. And this is a straightforward method but this is going to be very slow imagine the wealth, there's a lot of documents. If you do this then it will take a long time to answer your query. So the question now is why would the invert index be much faster? Well it has to do is the word distribution in text. So, here's some common phenomena of word distribution in the text. There are some languages independent of patterns that seem to be stable.

우리가 용어 A와 일치하는 모든 문서를 간단히 가져오고 용어 B와 일치하는 모든 문서를 가져 왔기 때문에 분명히 알 수 있습니다. 그런 다음 A와 B 같은 쿼리에 응답하기 위해 교차로를 가져옵니다. 또는 노동 조합이 질의 A 또는 B에 응답하도록하십시오. 이렇게하면 대답하기가 매우 쉽습니다. 그것은 매우 빨리 될 것입니다. 이제 다단계 키워드 쿼리는 어떻게됩니까? 예를 들어 벡터 공간 모델에 대해 이야기하고 문서와 함께 이러한 쿼리를 수행하고 점수를 생성합니다. 점수는 집계 된 용어 가중치를 기반으로합니다. 따라서이 경우 부울 쿼리가 아니지만 비슷한 방식으로 채점을 실제로 수행 할 수 있습니다. 기본적으로 disjunctive Boolean 쿼리와 비슷합니다. 기본적으로 A 또는 B와 같습니다. 적어도 하나의 쿼리 용어와 일치하는 모든 문서를 통합 한 다음 용어 가중치를 집계합니다. 따라서 이것은 일반적으로 문서 채점을 위해 역 색인을 사용하는 기본적인 아이디어입니다. 그리고 우리는 나중에 이것에 대해 더 자세히 이야기 할 것입니다. 그러나 지금은 왜 두 지수가 모두 좋은 아이디어인지 질문을 해봅시다. 기본적으로 문서를 순차적으로 스캔하는 것보다 효율적입니다. 이것은 명백한 접근법입니다. 각 문서의 점수 만 계산하면 정렬 할 수 있습니다. 그리고 이것은 간단한 방법입니다. 그러나 이것은 매우 천천히 진행될 것입니다. 부를 상상해보십시오. 많은 문서가 있습니다. 이렇게하면 쿼리에 응답하는 데 오랜 시간이 걸립니다. 이제 질문은 왜 반전 지수가 훨씬 더 빠를까요? 글의 단어 배포가 필요합니다. 그래서, 여기에 텍스트에서 단어 배포의 일반적인 현상이 있습니다. 안정적으로 보이는 패턴과 독립적 인 언어가 있습니다.

**14:00**

And these patterns are basically characterized by the following pattern. A few words like the common words like the, a, or we occur very, very frequently in text. So they account for a large percent of occurrences of words.

그리고 이 패턴들은 기본적으로 다음과 같은 패턴을 특징으로합니다. the, a, 또는와 같은 일반적인 단어와 같은 몇 단어는 텍스트에서 매우 빈번하게 발생합니다. 그래서 그들은 단어의 큰 비율을 차지합니다.

**14:19**

But most words would occur just rarely. There are many words that occur just once, let's say, in a document or once in the collection. And there are many such. It's also true that the most frequent the words in one corpus they have to be rare in another. That means although the general phenomenon is applicable, was observed in many cases that exact words that are common may vary from context to context. So this phenomena is characterized by what's called a Zipf's Law. This law says that the rank of a word multiplied by the frequency of the word is roughly constant.

그러나 대부분의 단어는 드물게 발생합니다. 하나의 문서에서 또는 한 번 컬렉션에서 한 번만 발생하는 많은 단어가 있습니다. 그리고 그러한 것들이 많이 있습니다. 또한 한 코퍼스에서 가장 빈번하게 사용되는 단어는 다른 코퍼스에서 드문 경우 여야합니다. 이것은 일반적인 현상이 적용 가능 하긴하지만 많은 경우에 공통적 인 정확한 단어가 문맥마다 다를 수 있음을 의미합니다. 그래서이 현상은 지프 (Zipf)의 법칙에 의해 특징 지워집니다. 이 법칙에 따르면 단어의 순위에 단어의 빈도를 곱한 값은 대략 일정합니다.

**15:07**

So formally if we use F(w) to denote the frequency, r(w) to denote the rank of a word. Then this is the formula. It basically says the same thing, just mathematical term. Where C is basically a constant and so, and there is also a parameter, alpha, that might be adjusted to better fit any empirical observations. So if I plot the word frequencies in sorted order, then you can see this more easily. The x axis is basically the word rank. This is r(w) and the y axis is word frequency or F(w). Now this curve shows that the product of the two is roughly the constant. Now if you look at these words, we can see They can be separated into three groups. In the middle, it's the intermediary frequency words. These words tend to occur quite in a few documents, but they are not like those most frequent words. And they are also not very rare.

그래서 공식적으로 F (w)를 사용하여 주파수를 나타낼 때 r (w)는 단어의 순위를 나타냅니다. 다음은 수식입니다. 그것은 기본적으로 수학적 용어 인 똑같은 것을 말합니다. C는 기본적으로 상수이므로, 경험적 관측에 더 잘 맞도록 조정될 수있는 매개 변수 α도 있습니다. 따라서 단어 빈도를 정렬 된 순서대로 플롯하면 더 쉽게 알 수 있습니다. x 축은 기본적으로 단어 순위입니다. 이것은 r (w)이고 y 축은 단어 빈도 또는 F (w)입니다. 이제이 커브는 두 제품의 결과가 대략 일정하다는 것을 보여줍니다. 이제이 단어들을 보면, 우리는 세 그룹으로 분리 될 수 있음을 알 수 있습니다. 중간에, 그것은 중개자의 빈도 단어입니다. 이 단어들은 몇 가지 문서에서 상당히 나타나는 경향이 있지만, 가장 빈번한 단어와는 다릅니다. 그리고 그들은 또한 아주 희소하지도 않습니다.

**16:18**

So they tend to be often used in

**16:22**

queries and they also tend to have high TF-IDF weights. These intermediate frequency words. But if you look at the left part of the curve,

**16:35**

these are the highest frequency words. They are covered very frequently. They are usually words, like the, we, of Etc. Those words are very, very frequent and they are in fact the two frequent to be discriminated, and they are generally not very useful for retrieval. So they are often removed and this is called the stop words removal. So you can use pretty much just the kind of words in the collection to kind of infer what words might be stop words. Those are basically the highest frequency words.

그래서 그들은 종종 질의에 사용되는 경향이 있으며 TF-IDF 가중치가 높은 경향이 있습니다. 이러한 중간 주파수 단어. 그러나 곡선의 왼쪽 부분을 보면 가장 높은 빈도 단어입니다. 그들은 매우 자주 다룹니다. 그들은 일반적으로 우리 같은 단어입니다. 그 단어는 매우 빈번하며 실제로 두 단어가 차별을 당하기 쉽습니다. 검색에 유용하지는 않습니다. 그래서 그들은 종종 제거되고 이것을 정지 단어 제거라고합니다. 따라서 콜렉션에서 거의 모든 종류의 단어를 사용하여 어떤 단어가 정지 단어 일 수 있는지 추측 할 수 있습니다. 그것들은 기본적으로 가장 높은 빈도 단어입니다.

**17:13**

And they also occupy a lot of space in the inverted index. You can imagine the posting entries for such a word would be very long. And then therefore, if you can remove such words you can save a lot of space in the inverted index.

**17:29**

We also show the tail part, which has a lot of rare words. Those words don't occur very frequently, and there are many such words.

**17:39**

Those words are actually very useful for search also, if a user happens to be interested in such a topic. But because they're rare, it's often true that users aren't necessarily interested in those words. But retain them would allow us to match such a document accurately. They generally have very high IDF.

그리고 그들은 또한 역 색인에서 많은 공간을 차지합니다. 그러한 단어에 대한 게시물 항목이 매우 길다는 것을 상상할 수 있습니다. 따라서 그런 단어를 삭제할 수 있다면 거꾸로 된 색인에 많은 공간을 절약 할 수 있습니다. 드문 단어가 많은 꼬리 부분도 보여줍니다. 그 단어는 자주 발생하지 않으며 그러한 단어가 많이 있습니다. 사용자가 그러한 주제에 관심이있는 경우 이러한 단어는 실제로 검색에도 매우 유용합니다. 그러나 이들이 희귀하기 때문에 사용자가 반드시 해당 단어에 관심이없는 것은 사실입니다. 그러나 그것들을 유지하면 그러한 문서를 정확하게 일치시킬 수 있습니다. 그들은 일반적으로 IDF가 매우 높습니다.

**18:05**

So what kind of data structures should we use to store inverted index? Well, it has two parts, right. If you recall, we have a dictionary and we also have postings. The dictionary has modest size, although for the web it's still going to be very large but compare it with postings it's more distinct.

**18:26**

And we also need to have fast random access to the entries because we're going to look up on the query term very quickly. So therefore, we'd prefer to keep such a dictionary in memory if it's possible. If the collection is not very large, this is feasible, but if the collection is very large then it's in general not possible. If the vocabulary size is very large, obviously we can't do that. So, in general that's how it goes. So the data structures that we often use for storing dictionary, it would be direct access. There are structures like hash table, or b-tree if we can't store everything in memory or use disk. And then try to build a structure that would allow it to quickly look up entries.

따라서 역 색인을 저장하기 위해 어떤 종류의 데이터 구조를 사용해야합니까? 음, 두 부분으로되어 있습니다. 회상하면 사전이 있고 게시도 있습니다. 사전은 겸손한 크기이지만 웹의 경우 여전히 크기가 크지 만 게시와 비교하면 더 뚜렷합니다. 그리고 우리는 질의어를 매우 빨리 찾아 볼 것이기 때문에 엔트리에 대한 빠른 랜덤 액세스가 필요합니다. 그러므로 가능하다면 그러한 사전을 메모리에 보관하는 것을 선호합니다. 컬렉션이 너무 크지 않은 경우 가능하지만 컬렉션이 매우 큰 경우 일반적으로 불가능합니다. 어휘 크기가 매우 큰 경우에는 분명히 할 수 없습니다. 그래서, 일반적으로 그렇게됩니다. 따라서 사전을 저장하기 위해 자주 사용하는 데이터 구조는 직접 액세스가됩니다. 모든 것을 메모리에 저장하거나 디스크를 사용할 수 없다면 해시 테이블이나 b 트리와 같은 구조가 있습니다. 그런 다음 신속하게 항목을 조회 할 수있는 구조를 만들어보십시오.

**19:14**

For postings they are huge.

**19:18**

And in general, we don't have to have direct access to a specific entry. We generally would just look up a sequence of document IDs and frequencies for all the documents that matches the query term.

**19:33**

So would read those entries sequentially.

**19:37**

And therefore because it's large and we generally have store postings on disc, they have to stay on disc and they would contain information such as document IDs, term frequency or term positions, etcetera. Now because they are very large, compression is often desirable.

게시의 경우 거대합니다. 일반적으로 특정 항목에 직접 액세스 할 필요는 없습니다. 일반적으로 쿼리 용어와 일치하는 모든 문서에 대해 일련의 문서 ID와 빈도를 조회합니다. 따라서 해당 항목을 순차적으로 읽습니다. 그러므로 크기가 커서 우리는 일반적으로 디스크에 점포 포스팅을 가지고 있기 때문에 디스크에 머무르고 문서 ID, 용어 빈도 또는 용어 위치 등과 같은 정보를 포함해야합니다. 이제는 크기가 크기 때문에 압축하는 것이 바람직합니다.

**19:59**

Now this is not only to save disc space, and this is of course one benefit of compression, it It's not going to occupy that much space. But it's also to help improving speed.

이제는 디스크 공간을 절약 할 수있을뿐 아니라 물론 압축의 이점 중 하나입니다. 많은 공간을 차지하지는 않습니다. 그러나 속도 향상에도 도움이됩니다.

**20:13**

Can you see why? Well, we know that input and output would cost a lot of time. In comparison with the time taken by CPU. So, CPU is much faster but IO takes time and so by compressing the inverter index, opposing files will become smaller, and the entries, that we have the readings, and memory to process a query term, would be smaller, and then, so we can reduce the amount of tracking IO and that can save a lot of time. Of course, we have to then do more processing of the data when we uncompress the data in the memory. But as I said CPU is fast. So over all we can still save time.

왜 볼 수 있니? 우리는 입력과 출력에 많은 시간이 소요된다는 것을 알고 있습니다. CPU가 걸리는 시간과 비교. 따라서 CPU는 훨씬 빠르지 만 입출력에는 시간이 걸리므로 인버터 색인을 압축하면 반대 파일이 작아지고 읽기 항목이 있고 항목을 처리하는 항목이 작아집니다. 추적 입출력의 양을 줄여 많은 시간을 절약 할 수 있습니다. 물론 메모리의 데이터를 압축 해제 할 때 데이터를 더 처리해야합니다. 그러나 나는 CPU가 빠르다고 말했다. 그래서 우리는 여전히 시간을 절약 할 수 있습니다.

**21:08**

So compression here is both to save disc space and to speed up the loading of the index.

따라서 압축은 디스크 공간을 절약하고 색인로드 속도를 높입니다.

**2.5-System-Implementation---Inverted-Index-Construction**

**0:07**

This lecture is about the inverted index construction.

이 강의는 역 색인 구조에 관한 것입니다.

**0:13**

In this lecture, we will continue the discussion of system implementation. In particular, we're going to discuss how to construct the inverted index.

이 강의에서는 시스템 구현에 대한 논의를 계속할 것입니다. 특히 역 색인 생성 방법에 대해 논의 할 것입니다.

**0:25**

The construction of the inverted index is actually very easy if the dataset is very small. It's very easy to construct a dictionary and then store the postings in a file.

데이터 집합이 매우 작 으면 역 색인 생성은 실제로 매우 쉽습니다. 사전을 작성한 다음 파일에 게시물을 저장하는 것은 매우 쉽습니다.

**0:36**

The problem is that when our data is not able to fit to the memory then we have to use some special method to deal with it.

문제는 우리의 데이터가 메모리에 적합하지 않을 때 우리는 그것을 다루기 위해 특별한 방법을 사용해야한다는 것입니다.

**0:46**

And unfortunately, in most retrieval applications the dataset will be large. And they generally cannot be loaded into memory at once.

그리고 유감스럽게도 대부분의 검색 응용 프로그램에서는 데이터 세트가 커집니다. 그리고 그들은 일반적으로 즉시 메모리에로드 될 수 없습니다.

**0:56**

And there are many approaches to solve that problem, and sorting-based method is quite common and works in four steps as shown here. First, you collect the local termID, documentID and frequency tuples. Basically you will locate the terms in a small set of documents. And then once you collect those accounts you can sort those count based on terms. So that you will be able to local a partial inverted index and these are called rounds. And then you write them into a temporary file on the disk and then you merge in step 3. Do pairwise merging of these runs, until you eventually merge all the runs and generate a single inverted index.

그리고 그 문제를 해결하기위한 많은 접근법이 있으며 정렬 기반 방법은 매우 일반적이며 여기에 표시된 것처럼 네 단계로 작동합니다. 먼저 로컬 termID, documentID 및 빈도 튜플을 수집합니다. 기본적으로 작은 일련의 문서에서 용어를 찾습니다. 그런 다음 해당 계정을 수집하면 용어를 기준으로 개수를 정렬 할 수 있습니다. 그래서 부분 반전 된 색인을 지역화 할 수 있고 이것들을 라운드라고합니다. 그런 다음 디스크의 임시 파일에 기록한 다음 3 단계에서 병합합니다. 모든 실행을 병합하고 반전 된 단일 색인을 생성 할 때까지 이러한 실행을 쌍으로 병합하십시오.

**1:47**

So this is an illustration of this method. On the left you see some documents and on the right we have a term lexicon and a document ID lexicon. These lexicons are to map string-based representations of document IDs or terms into integer representations or map back from integers to the stream representation. The reason why we want our interest using integers to present these IDs is because integers are often easier to handle. For example, integers can be used as index for array, and they are also easy to compress.

그래서 이것은이 방법의 실례입니다. 왼쪽에는 몇 가지 문서가 있고 오른쪽에는 용어 어휘와 문서 ID 어휘집이 있습니다. 이러한 사전은 문서 ID 또는 용어의 문자열 기반 표현을 정수 표현으로 매핑하거나 정수에서 스트림 표현으로 매핑하는 것입니다. 우리가 정수를 사용하여이 ID를 제시하고자하는 이유는 정수가 종종 다루기 쉽기 때문입니다. 예를 들어 정수는 배열의 인덱스로 사용할 수 있으며 압축하기도 쉽습니다.

**2:34**

So this is one reason why we tend to map these strings into integers,

**2:42**

so that we don't have to carry these strings around. So how does this approach work? Well, it's very simple. We're going to scan these documents sequentially and then parse the documents and count the frequencies of terms. And in this stage we generally sort the frequencies by document IDs, because we process each document sequentially. So we'll first encounter all the terms in the first document. Therefore the document IDs are all ones in this case. And this will be followed by document IDs two and they are natural results in this only just because we process the data in a sequential order. At some point, we will run out of memory and that would have to write them into the disc. Before we do that we 're going to sort them, just use whatever memory we have. We can sort them and then this time we're going to sort based on term IDs. Note that here, we're using the term IDs as a key to sort. So all the entries that share the same term would be grouped together. In this case, we can see all the IDs of documents that match term 1 would be grouped together. And we're going to write this into that this is a temporary file. And would that allows you to use the memory to process and makes a batch of documents. And we're going to do that for all the documents. So we're going to write a lot of temporary files into the disc. And then the next stage is we do merge sort basically. We're going to merge them and then sort them. Eventually, we will get a single inverted index, where the entries are sorted based on term IDs.

이것이 우리가 이러한 문자열을 정수로 매핑하는 경향이있는 한 가지 이유입니다. 따라서이 문자열을 전달할 필요가 없습니다. 그러면이 접근 방식은 어떻게 작동합니까? 아주 간단합니다. 우리는이 문서를 순차적으로 스캔 한 다음 문서를 파싱하고 용어의 빈도를 세웁니다. 그리고이 단계에서는 각 문서를 순차적으로 처리하기 때문에 일반적으로 문서 ID로 빈도를 정렬합니다. 먼저 첫 번째 문서의 모든 용어를 접하게 될 것입니다. 따라서이 경우 문서 ID는 모두 1입니다. 그리고이 다음에 문서 ID 2가 뒤따를 것이고 순차적 인 순서로 데이터를 처리하기 때문에 이것은 단지 자연스러운 결과입니다. 어떤 시점에서, 우리는 메모리가 부족하고 디스크에 기록해야합니다. 우리가 그것을하기 전에 우리는 그들을 분류 할 것이고, 우리가 가지고있는 기억을 사용하십시오. 정렬 할 수 있으며 이번에는 용어 ID를 기준으로 정렬 할 것입니다. 여기에서는 정렬이라는 키로 ID라는 용어를 사용합니다. 따라서 동일한 용어를 공유하는 모든 항목은 함께 그룹화됩니다. 이 경우 용어 1과 일치하는 문서의 모든 ID가 함께 그룹화 된 것을 볼 수 있습니다. 그리고 우리는 이것을 임시 파일이라고 씁니다. 그러면 메모리를 사용하여 문서를 처리하고 배치 할 수 있습니다. 그리고 우리는 모든 문서에 대해 그렇게 할 것입니다. 그래서 우리는 많은 임시 파일을 디스크에 씁니다. 그리고 나서 다음 단계는 기본적으로 정렬을 병합하는 것입니다. 우리는 그들을 합쳐서 분류 할 것입니다. 결국 엔트리가 용어 ID를 기준으로 정렬되는 역전 된 단일 색인이 생성됩니다.

**4:46**

And on the top, we're going to see these are the older entries for the documents that match term ID 1. So this is basically, how we can do the construction of inverted index. Even though the data cannot be all loaded into the manner. Now, we mention earlier that because of hostings are very large, it's desirable to compress them. So let's now take a little bit how we compressed inverted index. Well the idea of compression in general, is for leverage skewed distributions of values. And we generally have to use variable-length encoding, instead of the fixed-length encoding as we use by default in a program manager like C++. And so how can we leverage the skewed distributions of values to compress these values? Well in general, we will use few bits to encode those frequent words at the cost of using longer bit string code those rare values. So in our case, let's think about how we can compress the TF, tone frequency.

맨 위에는 용어 ID 1과 일치하는 문서의 이전 항목이 표시됩니다. 기본적으로 역 색인 생성은 어떻게 할 수 있습니까? 데이터가 모든 방식으로로드 될 수는 없지만. 이제는 호스팅이 매우 크기 때문에 압축하는 것이 바람직합니다. 이제 역 색인을 어떻게 압축했는지 조금 봅시다. 일반적으로 압축의 개념은 왜곡 된 값 분포를 활용하는 것입니다. 그리고 우리는 C ++와 같은 프로그램 관리자에서 기본적으로 사용하는 고정 길이 인코딩 대신에 일반적으로 가변 길이 인코딩을 사용해야합니다. 그렇다면 이러한 값을 압축하기 위해 값의 비뚤어진 분포를 어떻게 활용할 수 있습니까? 일반적으로, 우리는 희소 가치가 더 긴 비트 문자열 코드를 사용하는 대신 자주 단어를 인코딩하기 위해 비트를 거의 사용하지 않을 것입니다. 그래서 우리의 경우, TF, 톤 주파수를 어떻게 압축 할 수 있는지 생각해 봅시다.

**6:05**

Now, if you can picture what the inverted index look like, and you will see in post things, there are a lot of tone frequencies. Those are the frequencies of terms in all those documents. Now, if you think about it, what kind of values are most frequent there? You probably will be able to guess that small numbers tend to occur far more frequently than large numbers. Why? Well, think about the distribution of words and this is to do the sip of slopes, and many words occur just rarely so we see a lot of small numbers. Therefore, we can use fewer bits for the small, but highly frequent integers and that's cost of using more bits for larger integers.

이제 역 색인이 어떻게 보이는지 알 수 있고 포스트 사물에서 보게되면 많은 톤 주파수가 있습니다. 그것들은 모든 문서에서 용어의 빈도입니다. 자, 당신이 그것에 대해 생각한다면, 어떤 종류의 가치가 가장 빈번합니까? 작은 숫자는 큰 숫자보다 훨씬 자주 발생하는 경향이있을 것입니다. 왜? 음, 단어의 분포에 대해 생각해보십시오. 이것은 경사면의 한 모금을하는 것입니다. 많은 단어가 거의 발생하지 않기 때문에 많은 작은 숫자가 보입니다. 따라서 작지만 자주 사용되는 정수는 더 적은 비트를 사용할 수 있으며 더 큰 정수에는 더 많은 비트를 사용하는 데 드는 비용입니다.

**6:58**

This is a trade off of course. If the values are distributed to uniform, then this won't save us any space, but because we tend to see many small values, they are very frequent. We can save on average even though sometimes when we see a large number we have to use a lot of bits.

이것은 물론 거래입니다. 값이 uniform으로 분산되면 공간을 절약 할 수 없지만 많은 작은 값을 보게되므로 빈번하게 발생합니다. 때때로 우리가 많은 숫자를 볼 때 우리는 많은 비트를 사용해야 만 평균적으로 저장할 수 있습니다.

**7:19**

What about the document IDs that we also saw in postings? Well they are not distributed in the skewed way. So how can we deal with that? Well it turns out that we can use a trick called a d-gap and that is to store the difference of these term IDs. And we can imagine if a term has matched that many documents then there will be longest of document IDs. So when we take the gap, and we take the difference between adjacent document IDs, those gaps will be small. So again, see a lot of small numbers. Whereas if a term occurred in only a few documents, then the gap would be large, the large numbers would not be frequent. So this creates some skewed distribution, that would allow us to compress these values.

게시물에서 본 문서 ID는 어떻게됩니까? 그럼 그들은 비뚤어진 방식으로 배포되지 않습니다. 그러면 우리는 어떻게 대응할 수 있습니까? 우리는 d-gap이라고하는 트릭을 사용할 수 있으며이 용어 ID의 차이점을 저장하는 것으로 나타났습니다. 그리고 용어가 많은 문서와 일치하면 가장 긴 문서 ID가있을 것이라는 것을 상상할 수 있습니다. 그래서 우리가 갭을 잡아서 인접한 문서 ID의 차이를 취하면 그 갭은 작아 질 것입니다. 다시 한번, 많은 작은 숫자들을보십시오. 용어가 소수의 문서에서만 발생했다면 그 격차는 클 것이고 큰 숫자는 빈번하지 않을 것입니다. 그래서 이것은 왜곡 된 분포를 만들어서 우리가이 값들을 압축 할 수있게합니다.

**8:11**

This is also possible because in order to uncover or uncompress these document IDs, we have to sequentially process the data. Because we stored the difference and in order to recover the exact document ID we have to first recover the previous document ID. And then we can add the difference to the previous document ID to restore the current document ID. Now this was possible because we only needed to have sequential access to those document IDs. Once we look up the term, we look up all the document IDs that match the term, then we sequentially process them. So it's very natural, that's why this trick actually works.

이러한 문서 ID를 확인하거나 압축을 풀려면 순차적으로 데이터를 처리해야하기 때문에 가능합니다. 차이점을 저장 했으므로 정확한 문서 ID를 복구하려면 먼저 이전 문서 ID를 복구해야합니다. 그런 다음 이전 문서 ID에 차이를 추가하여 현재 문서 ID를 복원 할 수 있습니다. 이제는 이러한 문서 ID에 순차적으로 액세스해야하기 때문에 가능했습니다. 용어를 찾으면 용어와 일치하는 모든 문서 ID를 찾은 다음 순차적으로 처리합니다. 그래서 매우 자연 스럽습니다. 그래서이 트릭이 실제로 작동합니다.

**8:53**

And there are many different methods for encoding. So binary code is a commonly used code in just any program language. We use basically fixed glance in coding. Unary code, gamma code, and delta code are all possibilities and there are many other possibilities. So let's look at some of them in more detail. Binary coding is really equal length coding, and that's a property for randomly distributed values. The unary coding is a variable length in coding method. In this case, integer this 1 will be encoded as x -1, 1 bit followed by 0. So for example, 3 will be encoded as 2, 1s followed by 0, whereas 5 will be encoded as 4, 1s, followed by 0, etc. So now you can imagine how many bits do we have to use for a large number like 100? So how many bits do you have to use exactly for a number like 100? Well exactly, we have to use 100 bits. So it's the same number of bits as the value of this number. So this is very inefficient if you were likely to see some large numbers. Imagine if you occasionally see a number like 1,000, you have to use 1,000 bits. So this only works well if you are absolutely sure that there will be no large numbers, mostly very often you see very small numbers. Now, how do you decode this code? Now since these are variable length encoding methods, you can't just count how many bits and then just stop.

그리고 인코딩에는 여러 가지 다른 방법이 있습니다. 따라서 바이너리 코드는 모든 프로그램 언어에서 일반적으로 사용되는 코드입니다. 기본적으로 코딩시 고정 된 모양을 사용합니다. 단항 코드, 감마 코드 및 델타 코드는 모두 가능하며 다른 많은 가능성이 있습니다. 그럼 좀 더 자세히 살펴 보겠습니다. 이진 코딩은 실제로는 길이 코딩과 동일하며 무작위로 분산 된 값을위한 속성입니다. 단항 코딩은 코딩 방법에서 가변 길이이다. 이 경우 정수 1은 x -1로 인코딩되고 1 비트 뒤에는 0이옵니다. 예를 들어, 3은 2, 1은 0, 5는 4, 1s, 0으로 인코딩됩니다. 등등. 그래서 지금 당신은 얼마나 많은 비트를 우리가 100과 같이 많은 수에 사용해야하는지 생각할 수 있습니다. 그러면 100과 같은 숫자에 대해 정확히 몇 비트를 사용해야합니까? 정확히 정확히 100 비트를 사용해야합니다. 따라서이 숫자 값과 동일한 비트 수입니다. 따라서 많은 수의 숫자를 보게 될 경우 매우 비효율적입니다. 가끔 1,000과 같은 숫자를 보게되면 1,000 비트를 사용해야한다고 상상해보십시오. 이것은 큰 숫자가 없을 것이라는 것을 절대적으로 확신하는 경우에만 효과적입니다. 주로 매우 자주 숫자를 보게됩니다. 자, 어떻게이 코드를 디코 드합니까? 이제 이들은 가변 길이 인코딩 방법이므로 몇 비트를 계산하고 멈출 수는 없습니다.

**10:38**

You can't say 8-bits or 32-bits, then you will start another code. They are variable length, so you will have to rely on some mechanism. In this case for unary, you can see it's very easy to see the boundary. Now you can easily see 0 would signal the end of encoding. So you just count up how many 1s you have seen and at the end you hit 0. You have finished one number, you will start another number.

8 비트 또는 32 비트라고 말할 수 없으면 다른 코드를 시작할 것입니다. 가변 길이이므로 일부 메커니즘에 의존해야합니다. 이 단항의 경우 경계를 쉽게 볼 수 있습니다. 이제 0으로 인코딩의 끝을 알 수 있습니다. 그래서 당신은 당신이 얼마나 많은 1을 보았는지를 세고 결국 0을 치게됩니다. 하나의 숫자를 마쳤 으면 다른 숫자를 시작할 것입니다.

**11:07**

Now we just saw that unary coding is too aggressive. In rewarding small numbers, and if you occasionally can see a very big number, it would be a disaster. So what about some other less aggressive method? Well gamma coding's one of them and in this method we can use unary coding for a transform form of that. So it's 1 plus the floor of log of x. So the magnitude of this value is much lower than the original x. So that's why we can afford using unary code for that. And so first I have the unary code for coding this log of x. And this would be followed by a uniform code or binary code. And this basically the same uniform code, and binary code are the same. And we're going to use this coder to code the remaining part of the value of x. And this is basically precisely x-1 to the floor of log of x

이제 단항 코딩이 너무 공격적이라는 것을 보았습니다. 보람있는 작은 숫자에서, 그리고 때때로 아주 큰 숫자를 볼 수 있다면 그것은 재앙입니다. 덜 공격적인 방법은 어떨까요? 좋은 감마 코딩은 그 중 하나이며, 이 방법에서는 변환 형식을 단항 코딩으로 사용할 수 있습니다. 그래서 x의 로그의 바닥에 1을 더한 것입니다. 따라서이 값의 크기는 원본 x보다 훨씬 작습니다. 이것이 우리가 단항 코드를 사용할 여력이있는 이유입니다. 그래서 먼저 x의 로그를 코딩하는 단항 코드가 있습니다. 그리고 이것은 균일 한 코드 또는 바이너리 코드에 의해 뒤 따른다. 그리고 이것은 기본적으로 동일한 유니 코드와 바이너리 코드가 같습니다. 이 코더를 사용하여 x 값의 나머지 부분을 코딩합니다. 그리고 이것은 기본적으로 x의 로그의 바닥까지 정확히 x-1입니다.

**12:25**

So the unary code are basically called the flow of log of x, well add one there and here. But the remaining part we'll be using uniform code through actually code the difference between the x and this 2 to the log of x.

**12:49**

And it's easy to show that for this

**12:55**

difference we only need to use up to this many bits and the floor of log of x bits.

따라서 단항 코드는 기본적으로 x의 로그 흐름이라 불리우며 여기에 하나를 추가하십시오. 하지만 우리는 균일 코드를 사용하여 나머지 부분을 실제로 x와 2의 차이를 x의 로그에 코드화합니다. 그리고이 차이에 대해 우리는이 많은 비트와 x 비트의 로그의 바닥까지 사용할 필요가 있음을 보여주기 쉽습니다.

**13:06**

And this is easy to understand, if the difference is too large, then we would have a higher floor of log of x.

그리고 이것은 쉽게 이해할 수 있습니다. 차이가 너무 크면 우리는 더 높은 X 로그를 갖게 될 것입니다.

**13:14**

So here are some examples for example, 3 is is encoded as 101. The first two digits are the unary code. So this isn't for the value 2, 10 encodes 2 in unary coding.

**13:32**

And so that means the floor of log of x is 1, because we won't actually use unary codes. In code 1 plus the flow of log of x, since this is two then we know that the flow of log of x is actually 1.

**13:52**

So that 3 is still larger than 2 to the 1. So the difference is 1, and the 1 is encoded here at the end.

**14:01**

So that's why we have 101 for 3. Now similarly 5 is encoded as 110, followed by 01.

예를 들어 여기에 몇 가지 예제가 있습니다. 3은 101로 인코딩됩니다. 처음 두 자리는 단항 코드입니다. 따라서 값 2는 아니며 10은 단항 코딩에서 2를 인코딩합니다. 그리고 이것은 x의 로그의 바닥이 1이라는 것을 의미합니다. 왜냐하면 우리는 실제로 단항 코드를 사용하지 않기 때문입니다. 코드 1에서 x의 로그 흐름을 더한 결과 x가 2이기 때문에 x의 로그 흐름이 실제로 1임을 알 수 있습니다. 따라서 3은 여전히 1보다 2보다 크므로 차이가 1이므로 1 여기에 결국 엔코딩됩니다. 그래서 우리는 3을 위해 101을 가지고 있습니다. 마찬가지로 5도 110으로 인코딩되고, 01로 인코딩됩니다.

**14:12**

And in this case the unary code in code 3. And so this is a unary code 110 and so the flow of log of x is 2. And that means we're going to compute a difference between 5 and the 2 to the 2 and that's 1. And so we now have again 1 at the end. But this time we're going to use 2 bits, because with this level of flow of log of x. We could have more numbers a 5, 6, 7 they would all share the same prefix here, 110. So in order to differentiate them, we have to use 2 bits in the end to differentiate them. So you can imagine 6 would be 10 here in the end instead of 01 after 10.

그리고 이 경우 코드 3의 단항 코드입니다. 그리고 이것은 단항 코드 110이므로 x의 로그 흐름은 2입니다. 즉, 5와 2의 차이를 2로 계산할 것입니다. 그것은 1입니다. 그래서 이제 우리는 마지막에 다시 1을가집니다. 하지만 이번에는 2 비트를 사용하려고합니다. 왜냐하면 x의 로그 수준에서. 우리는 5, 6, 7의 숫자를 더 많이 가질 수 있습니다. 110을 사용하면 서로 다른 접두사를 공유 할 수 있습니다. 따라서 이들을 구별하기 위해 결국 2 비트를 사용하여 구분해야합니다. 따라서 여러분은 6이 10 이후에 01 대신 끝에 10이 될 것이라고 상상할 수 있습니다.

**15:04**

It's also true that the form of a gamma code is always the first odd number of bits, and in the center there is a 0. That's the end of the unary code.

**15:18**

And before that or on the left side of this 0, there will be all 1s. And on the right side of this 0, it's binary coding or uniform coding.

감마 코드의 형식이 항상 첫 번째 홀수 비트이고 중심에 0이있는 것도 사실입니다. 이것이 단항 코드의 끝입니다. 그리고 그 전에 또는이 0의 왼쪽에는 1이 모두있을 것입니다. 그리고이 0의 오른쪽에 이진 코딩 또는 균일 코딩이 있습니다.

**15:32**

So how can you decode such code? Well you again first do unary coding. Once you hit 0, you have got the unary code and this also tell you how many bits you have to read further to decode the uniform code. So this is how you can decode a gamma code. There is also a delta code that's basically the same as a gamma code except that you replace the unary prefix with the gamma code. So that's even less conservative than gamma code in terms of wording the small integers. So that means, it's okay if you occasionally see a large number.

그런 코드를 어떻게 디코딩 할 수 있습니까? 그럼 다시 처음으로 단항 코딩을하십시오. 0을 치면 단항 코드가 생겨서 유니 코드를 해독하기 위해 더 읽어야하는 비트 수를 알 수 있습니다. 이것이 감마 코드를 디코딩 할 수있는 방법입니다. 단리 접두어를 감마 코드로 바꾸는 것을 제외하고 기본적으로 감마 코드와 동일한 델타 코드도 있습니다. 그래서 이것은 작은 정수를 표현할 때 감마 코드보다 덜 보수적입니다. 즉, 큰 숫자가 때때로 나타나는 경우에는 괜찮습니다.

**16:14**

It's okay with delta code.

**16:16**

It's also fine with the gamma code, it's really a big loss for unary code. And they are all operating of course, at different degrees of favoring short or favoring small integers. And that also means they would be appropriate for a sorting distribution. But none of them is perfect for all distributions. And which method works the best would have to depend on the actual distribution in your dataset. For inverted index compression, people have found that gamma coding seems to work well.

델타 코드는 괜찮습니다. 감마 코드도 괜찮습니다. 단항 코드의 경우 큰 손실입니다. 그리고 그들은 모두 짧은 과정을 선호하거나 작은 정수를 선호하는 정도의 차이로 모든 과정을 진행합니다. 그리고 그것은 또한 정렬 분배에 적합하다는 것을 의미합니다. 그러나 그들 중 누구도 모든 배포판에 완벽하지 않습니다. 그리고 가장 효과가 좋은 방법은 데이터 집합의 실제 배포본에 의존해야합니다. 역 색인 압축의 경우 사람들은 감마 코딩이 잘 작동하는 것으로 나타났습니다.

**16:55**

So how to uncompress inverted index? I will just talk about this. Firstly, you decode those encoded integers. And we just I think discussed the how we decode unary coding and gamma coding. What about the document IDs that might be compressed using d-gap? Well, we're going to do sequential decoding so supposed the encoded I list is x1, x2, x3 etc. We first decode x1 to obtain the first document ID, ID1. Then we can decode x2, which is actually the difference between the second ID and the first one. So we have to add the decoder value of x2 to ID1 to recover the value of the ID at this secondary position.

역 색인을 압축 해제하는 방법은 무엇입니까? 나는 이것에 대해서만 이야기 할 것이다. 먼저 인코딩 된 정수를 디코딩합니다. 그리고 우리는 단항 코딩과 감마 코딩을 어떻게 해독하는지에 대해 논의했습니다. d-gap을 사용하여 압축 될 수있는 문서 ID는 무엇입니까? 음, 우리는 순차적 인 디코딩을 할 것입니다. 그래서 인코딩 된 I리스트가 x1, x2, x3 등이라고 가정합니다. 먼저 x1을 디코딩하여 첫 번째 문서 ID 인 ID1을 얻습니다. 그러면 x2를 디코딩 할 수 있습니다. x2는 실제로 두 번째 ID와 첫 번째 ID의 차이입니다. 따라서이 2 차 위치에서 ID의 값을 복구하려면 x2의 디코더 값을 ID1에 추가해야합니다.

**17:46**

So this is where you can see the advantages of converting document IDs to integers. And that allows us to do this kind of compression. And we just repeat until we decode all the documents. Every time we use the document ID in the previous position to help to recover the document ID in the next position.

여기에서 문서 ID를 정수로 변환 할 때 얻을 수있는 이점을 확인할 수 있습니다. 그리고 그것은 우리가 이런 종류의 압축을 할 수있게 해줍니다. 그리고 우리는 모든 문서를 해독 할 때까지 반복합니다. 매번 우리는 이전 위치에서 문서 ID를 사용하여 다음 위치에서 문서 ID를 복구하는 데 도움을줍니다.

**2.6-System-Implementation---Fast-Search**

**0:14**

In this lecture, we're going to continue the discussion of system implementation. In particular, we're going to talk about how to support a faster search by using invert index.

이 강의에서는 시스템 구현에 대한 논의를 계속할 것입니다. 특히, 역 색인을 사용하여 더 빠른 검색을 지원하는 방법에 대해 논의 할 것입니다.

**0:26**

So let's think about what a general scoring function might look like.

**0:32**

Now of course, the vector space model is a special case of this, but we can imagine many other retrieval functions of the same form.

**0:42**

So the form of this function is as follows.

그럼 일반적인 점수 계산 함수가 어떻게 생겼는지 생각해 봅시다. 이제는 벡터 공간 모델이 특별한 경우이지만 동일한 형식의 다른 많은 검색 함수를 상상할 수 있습니다. 따라서 이 함수의 형식은 다음과 같습니다.

**0:46**

We see this scoring function of a document D and a query Q is defined as first a function of fa that adjustment a function that would consider two factors. That I'll assume here at the end, f sub d of d and f sub q of q. These are adjustment factors of a document and a query, so they are at the level of a document and the query. So and then inside of this function, we also see there's another function called h. So this is the main part of the scoring function and these as I just said of the scoring factors at the level of the whole document and the query. For example, document [INAUDIBLE] and this aggregate punching would then combine all these. Now inside this h function, there are functions that would compute the weights of the contribution of a matched query term ti.

**2:08**

So this g, the function g gives us the weight of a matched query term ti in document d.

**2:23**

And this h function would then aggregate all these weights. So for example, take a sum of all the matched query terms,

**2:36**

but it can also be a product or it could be another way of aggregating them.

우리는 문서 D의이 채점 함수를보고 질의 Q는 두 가지 요소를 고려하는 함수를 조정하는 fa의 함수로 정의됩니다. 마지막에 여기에서 가정 할 것이고, f와 d의 하위 집합과 q의 집합입니다. 이것은 문서와 쿼리의 조정 요소이므로 문서와 쿼리의 수준에 있습니다. 그래서이 함수의 내부에는 h라는 또 다른 함수가 있습니다. 그래서 이것은 전체 문서와 쿼리의 수준에서 채점하는 요소에 대해 말한 것처럼 채점 기능의 주요 부분입니다. 예를 들어 document [INAUDIBLE]와 이 집계 펀칭은이 모든 것을 결합합니다. 이제이 h 함수 내부에는 일치 된 쿼리 용어 ti의 기여도 가중치를 계산하는 함수가 있습니다. 그래서이 g, 함수 g는 문서 d에서 일치 된 질의어 ti의 가중치를 우리에게 준다. 그리고이 h 함수는이 모든 가중치를 합산합니다. 예를 들어, 일치하는 모든 쿼리 용어의 합계를 취하십시오. 그러나 제품이 될 수도 있고, 이를 집계하는 다른 방법 일 수도 있습니다.

**2:41**

And then finally, this adjustment the functioning would then consider the document level or query level factors to further adjust this score, for example, document [INAUDIBLE]. So, this general form would cover many state of [INAUDIBLE] functions. Let's look at how we can score documents with such a function using virtual index.

**3:07**

So, here's a general algorithm that works as follows. First this query level and document level factors can be pre-computed in the indexing time. Of course, for the query we have to compute it at the query time but for document, for example, document [INAUDIBLE] can be pre-computed. And then, we maintain a score accumulator for each document d to computer h.

그런 다음 마지막으로이 기능을 조정하면 문서 수준 또는 쿼리 수준 요소를 고려하여이 점수를 추가로 조정할 수 있습니다 (예 : document [INAUDIBLE]). 그래서,이 일반적인 형식은 [INAUDIBLE] 기능의 많은 상태를 다룰 것입니다. 가상 인덱스를 사용하여 이러한 함수로 문서를 채점하는 방법을 살펴 보겠습니다. 여기에 다음과 같이 작동하는 일반적인 알고리즘이 있습니다. 먼저이 쿼리 수준 및 문서 수준 요소는 인덱싱 시간에 미리 계산 될 수 있습니다. 물론 쿼리의 경우 쿼리 시간에 계산해야하지만 문서의 경우 [INAUDIBLE] 문서를 미리 계산할 수 있습니다. 그리고 나서, 우리는 컴퓨터에 각 문서 d에 대한 점수 누적기를 유지합니다.

**3:34**

An h is an aggregation function over all the matching query terms. So how do we do that? For each period term we're going to do fetch the inverted list from the invert index. This will give us all the documents that match this query term

**3:52**

and that includes d1, f1 and so dn fn. So each pair is a document ID and the frequency of the term in the document. Then for each entry d sub j and f sub j are particular match of the term in this particular document d sub j. We'll going to compute the function g that would give us something like weight of this term, so we're computing the weight completion of matching this query term in this document. And then, we're going to update the score accumulator for this document and this would allow us to add this to our accumulator that would incrementally compute function h. So this is basically a general way to allow pseudo computer or functions of this form by using the inbound index. Note that we don't have to attach any of document and that didn't match any query term. Well, this is why it's fast, we only need to process the documents that matched at least one query term. In the end, then we're going to adjust the score the computer this function f sub a and then we can sort. So let's take a look at a specific example. In this case, let's assume the scoring function is a very simple one, it just takes the sum of t f, the role of t f, the count of a term in the document.

h는 모든 일치하는 u 리 용어에 대한 집계 함수입니다. 그럼 우리가 어떻게해야합니까? 각 기간에 대해 역 색인에서 반전 된 목록을 가져 오려고합니다. 이렇게하면이 검색어와 일치하고 d1, f1 및 dn fn을 포함하는 모든 문서가 제공됩니다. 따라서 각 쌍은 문서 ID 및 문서의 용어 빈도입니다. 그런 다음 각 항목에 대해 d sub j 및 f sub j는이 특정 문서 d sub j에서 용어의 특정 일치입니다. 우리는이 용어의 가중치와 같은 것을주는 함수 g를 계산할 것이므로이 문서에서이 쿼리 용어와 일치하는 가중치 완성을 계산할 것입니다. 그리고 나서이 문서의 점수 누적기를 업데이트 할 것입니다. 그러면 누적 계산기에이를 추가하여 점진적으로 함수 h를 계산할 수 있습니다. 따라서 기본적으로 인바운드 인덱스를 사용하여 의사 (pseudo) 컴퓨터 또는이 양식의 기능을 허용하는 일반적인 방법입니다. 문서를 첨부 할 필요가 없으며 검색어가 일치하지 않습니다. 글쎄, 이것이 빠른 이유이기 때문에 적어도 하나의 쿼리 용어와 일치하는 문서 만 처리하면됩니다. 결과적으로 컴퓨터에서이 점수를 조정하고 점수를 매겨서 정렬 할 수 있습니다. 이제 구체적인 예를 살펴 보겠습니다. 이 경우 득점 함수가 매우 단순하다고 가정 해 보겠습니다.이 함수는 t f의 합계, t f의 역할, 문서의 용어 수를 취합니다.

**5:25**

This simplification would help shield the algorithm clearly. It's very easy to extend the computation to include other weights like the transformation of tf, or [INAUDIBLE] or IDF [INAUDIBLE]. So let's take a look at specific example, where the queries information security

**5:48**

and it show some entries of invert index on the right side. Information occurred in four documents and their frequencies are also there, security occurred in three documents. So let's see how the arrows works, so first we iterate overall query terms and we fetch the first query then, what is that? That's information, right? And imagine we have all these score accumulators who score the,

**6:17**

scores for these documents. We can imagine there will be other but then they will only be allocated as needed. So before we do any waiting of terms, we don't even need a score of. That comes actually we have these score accumulators eventually allocating.

이 단순화는 알고리즘을 명확하게 보호하는 데 도움이됩니다. tf 또는 [INAUDIBLE] 또는 IDF [INAUDIBLE]의 변환과 같은 다른 가중치를 포함하도록 계산을 확장하는 것은 매우 쉽습니다. 이제 특정 예제를 살펴 보겠습니다. 여기서 쿼리 정보 보안과 오른쪽에 반전 인덱스 항목이 표시됩니다. 정보는 네 개의 문서에서 발생했으며 그 빈도도 있으며 세 가지 문서에서 보안이 발생했습니다. 이제 화살표가 어떻게 작동하는지 살펴 보겠습니다. 먼저 전체 쿼리 용어를 반복하고 첫 번째 쿼리를 가져옵니다. 그게 무엇입니까? 그게 정보 야, 맞지? 그리고이 모든 점수 누적 점수를 가진 점수 기록자가 있다고 가정 해보십시오. 우리는 다른 사람들이있을 것이라고 상상할 수 있지만, 필요할 때만 배정 될 것입니다. 그래서 우리가 조건을 기다리기 전에, 우리는 심지어 점수를 필요로하지 않습니다. 사실 우리는 결국 점수 누적기를 결국 할당하게됩니다.

**6:38**

So lets fetch the interest from the entity [INAUDIBLE] for information, that the first one.

**6:46**

So these four accumulators obviously would be initialize as zeros.

따라서 엔티티 [INAUDIBLE]가 첫 번째 정보에 대한 관심을 가져올 수 있습니다. 따라서이 4 개의 누산기는 분명히 0으로 초기화됩니다.

**6:51**

So, the first entry is d1 and 3, 3 is occurrences of information in this document. Since our scoring function assume that the score is just a sum of these raw counts. We just need to add a 3 to the score accumulator to account for the increase of score due to matching this term information, a document d1. And then, we go to the next entry, that's d2 and 4 and then we add a 4 to the score accumulator of d2. Of course, at this point, that we will allocate the score accumulator as needed. And so at this point, we allocated the d1 and d2, and the next one is d3, and we add one, we allocate another score [INAUDIBLE] d3 and add one to it. And then finally, the d4 gets a 5, because the term information occurred five times in this document. Okay, so this completes the processing of all the entries in the invert index for information. It processed all the contributions of matching information in this four documents.

따라서 첫 번째 항목은 d1과 3이며, 3은이 문서에서 정보가 표시됩니다. 우리의 채점 함수는 점수가이 날 카운트의 합계라고 가정하기 때문에. 이 용어 정보와 일치하는 점수의 증가를 설명하기 위해 점수 누적 계산기에 3을 추가하면됩니다. 문서 d1. 그런 다음 d2와 4의 다음 항목으로 이동 한 다음 d2의 점수 누적기에 4를 더합니다. 물론,이 시점에서 우리는 필요에 따라 점수 누적기를 할당 할 것입니다. 그래서이 시점에서 d1과 d2를 할당하고 다음 d3을 할당하고 하나를 더하고 다른 점수 [INAUDIBLE] d3을 할당하고 하나를 더합니다. 마지막으로 d4는 5라는 정보를 얻습니다. 왜냐하면 용어 정보가이 문서에서 5 번 발생했기 때문입니다. 좋아, 이렇게하면 정보에 대한 역 색인의 모든 항목 처리가 완료됩니다. 이 네 가지 문서에서 일치하는 정보의 모든 기여를 처리했습니다.

**8:01**

So now, our error will go to the next that's security. So, we're going to fetch all the inverted index entries for security.

이제는 오류가 보안 문제로 이어질 것입니다. 따라서 우리는 보안을 위해 모든 역 색인 항목을 가져올 것입니다.

**8:10**

So, in this case, there are three entries, and we're going to go through each of them. The first is d2 and 3 and that means security occur three humps in d2 and what do we do? Well, we do exactly the same, as what we did for information. So, this time we're going to change the score [INAUDIBLE] d2 since it's already allocated and what we do is we'll add 3 to the existing value which is a 4, so we now get a 7 for d2.

**8:41**

D2 score is increased because the match that falls the information and the security. Go to the next entry, that's d4 and 1, so we would the score for d4 and again, we add 1 to d4 so d4 goes from 5 to 6. Finally, we process d5 and a 3. Since we have not yet allocated a score accumulated for d5, at this point, we're going to allocate 1 for d5, and we're going to add a 3 to it. So, those scores, of the last rule, are the final scores for these documents.

따라서 이 경우 세 개의 항목이 있으며 각 항목을 살펴 보겠습니다. 첫 번째는 d2와 3이며 이는 보안이 d2에서 세 개의 혹이 발생한다는 것을 의미하며 우리는 무엇을합니까? 우리가 정보를 위해했던 것과 똑같은 일을합니다. 그래서, 이 시간 우리는 이미 할당 이후 점수 [들리지] (D2)를 변경하려고하고 우리가 할 것은 우리가 4 기존 값에 3을 추가 할 것입니다, 그래서 우리는 지금 (D2)에 대한 7을 얻을 수 있습니다.

**9:20**

If our scoring function is just a simple some of TF values.

우리의 채점 기능이 TF 값 중 일부일뿐입니다.

**9:27**

Now, what if we, actually, would like to do form addition? Well, we going to do the [INAUDIBLE] at this point, for each document.

**9:36**

So, to summarize this, all right, so you can see, we first process the information determine query term information and we processed all the entries in what index for this term. Then we process the security, all right, its worst think about what should be the order of processing here when we can see the query terms? It might make a difference especially if we don't want to keep all the score accumulators. Let's say, we only want to keep the most promising score accumulators. What do you think would be a good order to go through? Would you process a common term first or would you process a rare term first?

실제로 양식을 추가하고 싶다면 어떻게해야할까요? 우리는이 시점에서 각 문서에 대해 [무관심]을 수행 할 것입니다. 그래서 이것을 요약하면 알 수 있듯이 정보를 쿼리 정보 용어로 처리하고이 용어에 대한 색인의 모든 항목을 처리했습니다. 그런 다음 보안을 처리합니다. 좋습니다. 최악의 경우 쿼리 순서를 볼 때 처리 순서가 무엇인지 생각해보십시오. 우리가 모든 점수 누적기를 유지하고 싶지 않다면 특히 효과가있을 것입니다. 예를 들어, 우리는 가장 유망한 점수 누적기를 유지하기를 원합니다. 통과하기에 좋은 순서라고 생각하니? 먼저 공통 용어를 처리 하시겠습니까? 아니면 희귀 용어를 먼저 처리 하시겠습니까?

**10:24**

The answers is we just go to who should process the rare term first. A rare term would match a few documents, and then the score contribution would be higher, because the ideal value would be higher. And then, it allows us to attach the most diplomacy documents first. So, it helps pruning some non-promising ones, if we don't need so many documents to be returned to the user. So those are all heuristics for further improving the accuracy. Here you can also see how we can incorporate the idea of waiting, right? So they can [INAUDIBLE] when we incorporate [INAUDIBLE] when we process each query time. When we fetch the inverted index we can fetch the document frequency and then we can compute IDF. Or maybe perhaps the IDF value has already been precomputed when we indexed the documents. At that time, we already computed the IDF value that we can just fetch it, so all these can be done at this time. So that would mean when we process all the entries for information, these words would be adjusted by the same IDF, which is IDF for information.

대답은 드문 용어를 먼저 처리해야하는 사람에게로 이동하는 것입니다. 드문 용어는 몇 가지 문서와 일치 할 것이고 이상적인 값이 더 높기 때문에 점수 기여도가 더 높을 것입니다. 그리고 가장 먼저 외교 문서를 첨부 할 수 있습니다. 그래서 우리가 너무 많은 문서가 사용자에게 반환 될 필요가 없다면, 일부 유망하지 않은 것들을 가지 치기 (pruning) 도움이됩니다. 그래서 그것들은 정확성을 더욱 향상시키기위한 발견 적 방법입니다. 여기서 우리는 대기의 아이디어를 어떻게 통합 할 수 있는지도 볼 수 있습니다. 맞습니까? 따라서 우리는 각 질의 시간을 처리 할 때 [INAUDIBLE]을 통합 할 때 [INAUDIBLE] 할 수 있습니다. 역 색인을 가져올 때 문서 빈도를 가져온 다음 IDF를 계산할 수 있습니다. 아니면 우리가 문서를 색인 할 때 이미 IDF 값이 미리 계산되어있을 수도 있습니다. 그 당시 우리는 이미 가져올 수있는 IDF 값을 이미 계산 했으므로이 모든 것이 현재 수행 될 수 있습니다. 따라서 모든 정보 항목을 처리 할 때 동일한 단어 IDF (정보 IDF)가 이 단어를 조정하게됩니다.

**11:36**

So this is the basic idea of using inverted index for fast research and it works well for all kinds of formulas that are of the general form. And this generally, the general form covers actually most state of art retrieval functions. So there are some tricks to further improve the efficiency, some general techniques to encode the caching. This is we just store some results of popular queries, so that next time when you see the same query, you simply return the stored results. Similarly, you can also slow the list of inverted index in the memory for a popular term. And if the query term is popular likely, you will soon need to factor the inverted index for the same term again. So keeping it in the memory would help, and these are general techniques for improving efficiency. We can also keep only the most promising accumulators because a user generally doesn't want to examine so many documents. We only need to return high qualities subset of documents that likely are ranked on the top.

따라서 이것은 빠른 연구를 위해 역 색인을 사용하는 기본적인 아이디어이며 일반적인 형태의 모든 수식에 적합합니다. 그리고 이것은 일반적으로, 일반적인 형태는 사실 예술 검색 기능의 대부분의 상태를 포함합니다. 그래서 효율성을 향상시키기위한 몇 가지 트릭, 캐싱을 인코딩하는 몇 가지 일반적인 기술이 있습니다. 이것은 인기있는 쿼리 결과를 저장하기 때문에 다음에 같은 쿼리를 볼 때 저장된 결과 만 반환하면됩니다. 비슷하게 인기있는 용어의 경우 메모리에서 역 색인의 목록을 느리게 할 수도 있습니다. 그리고 검색어가 인기있는 경우 곧 같은 용어에 대한 역 색인을 다시 고려해야합니다. 따라서 그것을 기억에 유지하는 것이 도움이 될 것이며, 이는 효율성을 향상시키는 일반적인 기술입니다. 사용자가 일반적으로 너무 많은 문서를 검토하기를 원하지 않기 때문에 가장 유망한 축전기 만 유지할 수도 있습니다. 최상위 순위에 올 가능성이 높은 문서의 하위 집합만을 반환해야합니다.

**12:47**

For that purpose, we can then prune the accumulators. We don't have to store all the accumulators. At some point, we just keep the highest value accumulators. Another technique is to do parallel processing and that's needed for really process in such a large data set like the web data set. And you scale up to the Web-scale really to have the special techniques you do parallel processing and to distribute the storage of files on machines. So here is a list of some text retrieval toolkits, it's not a complete list. You can find more information at this URL on the bottom. And here, I listed your four here, Lucene's one of the most popular toolkits that can support a lot of applications and it has very nice support for applications. You can use it to build a search engine application very quickly. The downside is that it's not that easy to extend it, and the algorithms implemented they are also not the most advanced algorithms. Lemur or Indri is another toolkit that does not have such a nice support web application as Lucene but it has many advanced search algorithms and it's also easy to extend. Terrier is yet another toolkit that also has good support for application capability and some advanced algorithms. So that's maybe in between Lemur or Lucene, or maybe rather combining the strands of both, so that's also useful tool kit. MeTA is a toolkit that we will use for the problem assignment and this is a new toolkit that has

**14:47**

a combination of both text retrieval algorithms and text mining algorithms. And so talking models are implement they are a number of text analysis algorithms implemented in the toolkit as well as basic search algorithms. So to summarize all the discussion about the System Implementation,

**15:11**

here are the major takeaway points. Inverted index is the primary data structure for supporting a search engine and that's the key to enable faster response to a user's query.

그 목적을 위해, 우리는 누적기를 정리할 수 있습니다. 우리는 모든 축전지를 저장할 필요가 없습니다. 어떤 시점에서, 우리는 단지 가장 높은 값의 누적기를 유지합니다. 또 다른 기술은 병렬 처리를 수행하는 것이며 웹 데이터 세트와 같은 대규모 데이터 세트에서 실제로 처리하는 데 필요합니다. 또한 웹 스케일로 확장하여 병렬 처리를 수행하고 컴퓨터에 파일 저장 영역을 분산시키는 특별한 기술을 보유하게됩니다. 여기 몇몇 텍스트 검색 툴킷 목록이 있습니다. 완전한 목록은 아닙니다. 자세한 내용은 하단의이 URL을 참조하십시오. 여기 Lucene의 많은 네 가지 애플리케이션을 지원할 수있는 가장 인기있는 툴킷 중 하나 인 네 가지를 나열 했으므로 애플리케이션에 대한 지원이 매우 좋습니다. 이 도구를 사용하여 매우 빠르게 검색 엔진 응용 프로그램을 만들 수 있습니다. 단점은 확장하기가 쉽지 않으며 구현 된 알고리즘이 가장 진보 된 알고리즘이 아니라는 것입니다. Lemur 또는 Indri는 Lucene과 같은 훌륭한 지원 웹 응용 프로그램이없는 또 다른 툴킷이지만 많은 고급 검색 알고리즘이 있으며 확장하기도 쉽습니다. Terrier는 응용 프로그램 기능 및 일부 고급 알고리즘을 잘 지원하는 또 다른 툴킷입니다. 그래서 아마 Lemur 또는 Lucene 사이에있을 수도 있고, 아니면 양쪽 모두의 가닥을 결합하는 것일 수도 있으므로 유용한 도구 키트이기도합니다. MeTA는 문제 할당을 위해 사용할 툴킷이며, 텍스트 추출 알고리즘과 텍스트 마이닝 알고리즘을 결합한 새로운 툴킷입니다. 그리고 말하는 모델은 툴킷과 기본 검색 알고리즘에서 구현되는 수많은 텍스트 분석 알고리즘입니다. 따라서 시스템 구현에 대한 모든 논의를 요약하면 다음과 같습니다. 반전 된 색인은 검색 엔진을 지원하기위한 기본 데이터 구조이며 사용자 쿼리에보다 신속하게 응답 할 수있는 열쇠입니다.

**15:26**

And the basic idea is to preprocess the data as much as we can, and we want to do compression when appropriate. So that we can save disk space and we can speed up IO and processing of inverted index in general. We talked about how to construct the invert index when the data can't fit into the memory. And then we talk about faster search using that index basically, what's we exploit the invective index to accumulate a scores for documents [INAUDIBLE] algorithm. And we exploit the Zipf's law to avoid the touching many documents that don't match any query term and this algorithm can actually support a wide range of ranking algorithms.

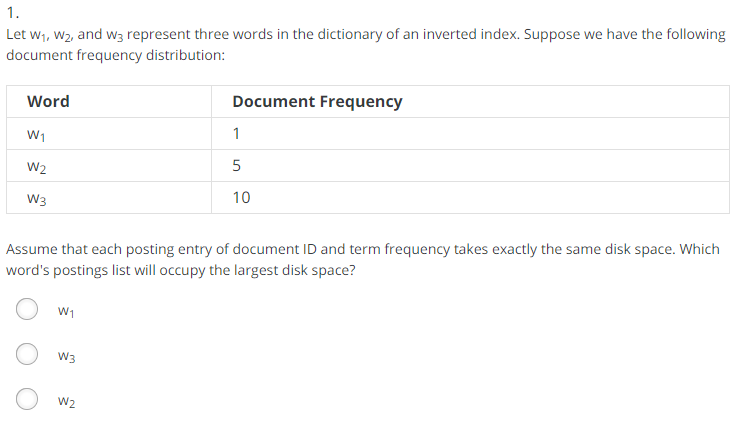
그리고 기본 아이디어는 가능한 한 많은 양의 데이터를 전처리하는 것이며 적절한 경우 압축을 수행하려고합니다. 그래서 우리는 디스크 공간을 절약 할 수 있고 전반적인 역 색인의 IO와 처리 속도를 높일 수 있습니다. 데이터가 메모리에 맞지 않을 때 역 색인을 만드는 방법에 대해 이야기했습니다. 그런 다음 기본적으로 색인을 사용하는 빠른 검색에 대해 이야기합니다. 문서의 [INAUDIBLE] 알고리즘 점수를 누적하는 비 활동 색인을 활용하는 것은 무엇입니까? 그리고 우리는 쿼리 용어와 일치하지 않는 많은 문서를 만지는 것을 피하기 위해 지프의 법칙을 이용합니다.이 알고리즘은 실제로 광범위한 순위 알고리즘을 지원할 수 있습니다.

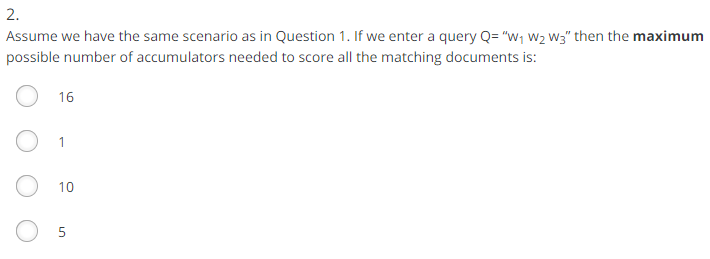
**16:13**

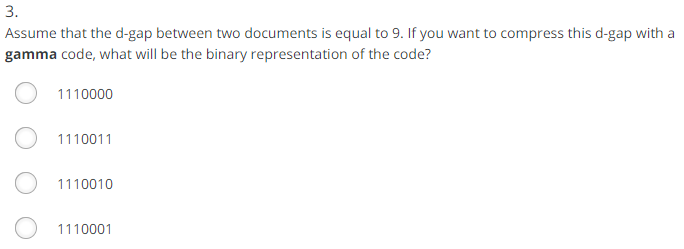
So these basic techniques have great potential for further scaling up using distributed file system, parallel processing, and caching. Here are two additional readings you can take a look, if you have time and you are interested in learning more about this. The first one is a classical textbook on the efficiency o inverted index and the compression techniques. And how to, in general feel that the efficient any inputs of the space, overhead and speed. The second one is a newer textbook that has a nice discussion of implementing and evaluating search engines.

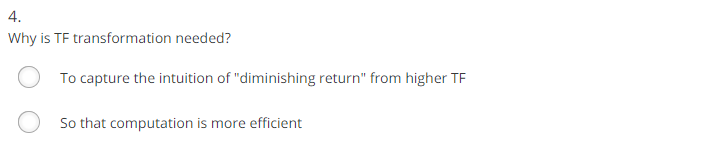
따라서 이러한 기본 기술은 분산 파일 시스템, 병렬 처리 및 캐싱을 사용하여 추가 확장이 가능합니다. 시간이 있고 이것에 대해 더 많은 것을 배우고 싶다면 두 가지 더 읽을 수 있습니다. 첫 번째는 역 색인 및 압축 기술의 효율성에 대한 고전 교과서입니다. 그리고 어떻게, 일반적으로 공간, 오버 헤드 및 속도의 효율적인 입력. 두 번째는 검색 엔진을 구현하고 평가하는 좋은 토론이있는 최신 교과서입니다.

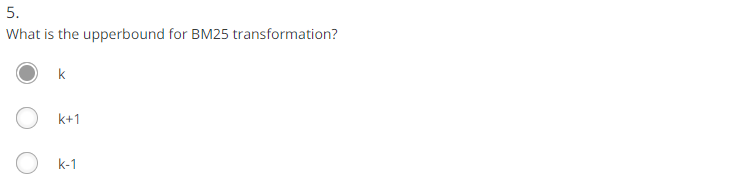
**Practice Quiz**

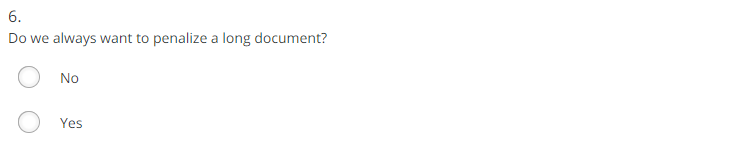


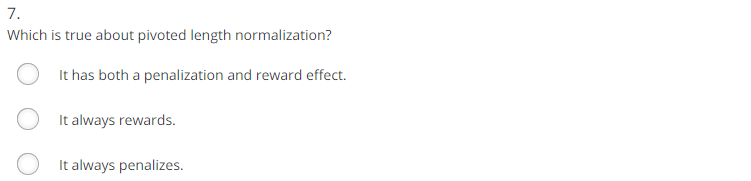


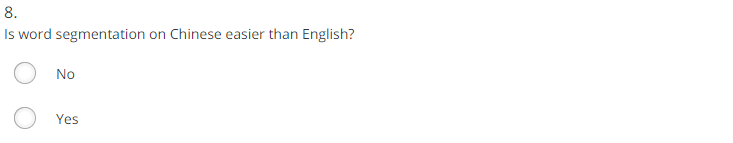


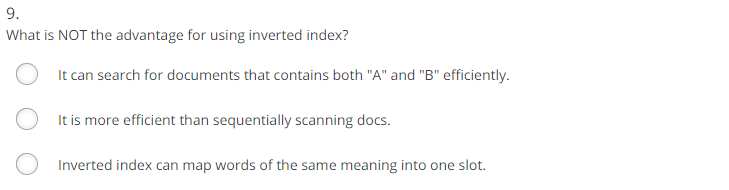


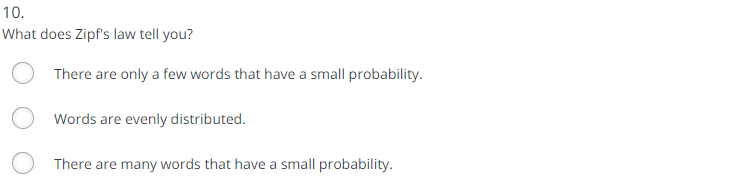












**Quiz**

