KRISTAL-II에서의 벡터 공간 모델 질의 모듈 구현

1998. 7.

김진숙

연구개발정보센터 정보지원부 검색지원실

1. 벡터 공간 모델(Vector Space Model) 고찰

표기

- d : 문서(document) 번호
- $f_{d,t}$: 문서내 단어 t의 출현 빈도(tf)
- f_t : 단어 t를 포함하는 문서의 총 수(df)
- M : 가중치 계산 (Measure)
- N : 문서 collection내의 총 문헌수
- n : 문서 collection내의 색인어 총수
- t : 단어(term)
- **TF*IDF** 규칙 : 문서 가중치를 단어빈도*역문헌빈도(**T**erm **F**requency * **I**nverse **D**ocument **F**requency)로 계산하는 규칙
- $w_{d,t}$: 문서-단어 가중치(document-term weight)
- $w_{q,t}$: 질의-단어 가중치(query-term weight)
- w_t : 단어 가중치(term weight)

d	Document D_d
1	apple balloon balloon elephant apple apple
2	Chocolate balloon balloon chocolate apple chocolate duck
3	Balloon balloon balloon balloon elephant balloon
4	Chocolate balloon elephant
5	Balloon apple chocolate balloon
6	Elephant elephant elephant chocolate elephant

표 1-1. 문서 모음 예제 (문서 6, 단어종류 5)

Inner product similarity

질의에 대한 문서의 중요성을 결정하는 하나의 방법으로 질의와 문서의 Inner product를 사용할 수 있다. 표 1-2(a)는 표 1-1에 나타난 문서들을 단어의 이진 벡터(binary vector)로 표시한 것이다. 표 1-2(b)는 질의를 이진 벡터로 표시하였다. 문서와 질의를 벡터로 표시했기 때문에 문서와 질의와의 관계를 벡터의 inner product로 표시할 수 있다. 예를 들어 질의와 문서 2의 유사도를 측정하면

$$M(\text{"chocolate,duck"}, D_2) = (0,0,1,1,0) \cdot (1,1,1,1,0) = 2.$$

로 계산될 것이다. 그러나 이 방법은 다음과 같은 여러 가지 문제점을 내포하고 있다.

1. 문서내의 단어빈도를 전혀 고려하지 않는다. 문서 D_6에서 *elephant*는 4번 출현하지만 1번

출현하는 것과 동일한 값을 산출한다.

2. 단어의 희소성을 고려하지 않는다. *duck*은 1개의 문서에서만 출현하고 *balloon*은 5개의 문서에서 출현하므로 문서 집합내에서 *duck*이 보다 중요한 단어라고 할 수 있으나 이에 대한 고려가 전혀 없다.
3. 문서가 길수록 질의어내의 질의어를 가지고 있을 확률이 높기 때문에, 긴 문서일수록 문서가중치가 높아질 확률이 높다.

첫번째 문제점은 문서 벡터를 이진 벡터로 표현하지 않고 단어 빈도의 벡터로 표현함으로써 해결할 수 있다. 이점을 고려하여 위의 유사도 측정을 다시 하면

$$M(\text{"chocolate,duck"}, D_2) = (0,0,1,1,0) \cdot (1,2,3,1,0) = 4$$

와 같아진다. 이를 일반화 시키면 질의와 문서의 유사도 측정은 다음과 같이 수식화 될 것이다.

$$M(Q, D_d) = Q \bullet D_d = \sum_{t=1}^{n} W_{q,t} \bullet W_{d,t}$$

(a)

d	단어 vector (a, b, c, d, e)
1	(1, 1, 0, 0, 1)
2	(1, 1, 1, 1, 0)
3	(0, 1, 0, 0, 1)
4	(0, 1, 1, 0, 1)
5	(1, 1, 1, 0, 0)
6	(0, 0, 1, 0, 1)

(b)

"duck"	(0, 0, 0, 1, 0)
"chocolate, duck"	(0, 0, 1, 1, 0)

표 1-2. 표 1-1에 대한 inner product 계산을 위한 벡터: (a)문서 벡터; (b) 질의 벡터 (a: apple, b: balloon, c: chocolate, d:duck, e: elephant)

두번째 문제점에 대해 생각해 보자. 문서 내에서 자주 출현하는 단어들의 경우 - 예를 들어 영어 문서내의 정관사 *the* - 흔하게 나타나지 않는 단어에 비해 중요성이 떨어진다고 할 수 있다. 반대로 흔하게 나타나지 않는 단어는 상대적으로 중요한 정보를 지닌다고 볼 수 있다. 이러한 관점에 따라 단어 t의 가중치 w_t를 단어의 출현 횟수 f_t와 총 문서의 수 N으로 나타내면 다음과 같다.

$$w_t = \log(N/f_t)$$

다시 문서 벡터는 단어 가중치 w_t와 단어 t의 문서내 출현 횟수 $f_{d,t}$로부터 아래와 같이 계산할 수 있다.

$$w_{d,t} = f_{d,t} * w_t = f_{d,t} * \log(N/f_t)$$

이와 같이 문서 가중치를 단어빈도*역문헌빈도 (**T**erm **F**requency * **I**nverse **D**ocument **F**requency)로 계산하는 방식을 **TF*IDF** 규칙이라고 한다.

표 1-1에서 단어 *duck*과 *elephant*의 예를 들어 보자. 단어 *duck*은 총 6개의 문서 중 1개의 문서에서 나타나므로

$$w_{duck} = \log(6/1) = 2.58$$

로 계산되고 단어 *elephant*는 6개의 문서 중 4개의 문서에서 나타나므로

$$w_{elephant} = \log(6/4) = 0.58$$

의 단어가중치를 가지게 된다. (코끼리 다섯 마리를 합쳐야 오리 한 마리의 무게가 나간다고나 할까?)

그러나 위와 같은 문서가중치 계산법에도 문제가 있다. 문서의 크기가 클수록 문서 D_d에 대한 $f_{d,t}$는 일반적으로 커질 것이다. 이를 해결하기 위해 문서의 길이가 가중치에 곱해지는 값을 정규화시킬 필요가 있다. 즉 위에서 계산된 가중치를 문서의 길이로 나누는 것이다.

$$M(Q, D_d) = \frac{1}{|D_d|} \sum_{t=1}^{n} W_{q,t} \bullet W_{d,t}$$

$$|D_d| = \sum_i f_{d,i}$$

여기서 문서 D_d의 길이는 문서 내에서 나타나는 색인어의 총 횟수를 의미한다.

벡터 공간 모델(Vector Space Model)

지금까지 논의된 바를 벡터 공간 모델을 이용하면 쉽게 정리가 된다. 질의마다 문서마다 서로 다른 길이를 가지고 있기 때문에 우리에게 의미가 있는 것은 질의와 문서의 길이 차이는 아니다. 실제로 필요한 것은 질의와 문서가 가지는 의미의 차이가 될 것이다. 달리 표현한다면 질의 벡터(vector)와 문서 벡터의 방향성의 차이, 즉 공간상의 각도의 차이로 표현할 수 있다.

질의 벡터와 문서 벡터의 공간상 각도 차이를 쉽게 이해할 수 있도록 벡터 공간 모델의 한 구현 방식인 코사인 측정법(cosine measure)을 통해 알아보자. 공간상의 각도차이를 코사인(cosine)을 이용하여 표현하면, 코사인은 방향이 같을 때 즉 서로간의 각도가 0일 때는 1, 서로 직각일 때는 0의 값을 가진다. 따라서 코사인의 값은 자체로 순위결정에 사용할 수 있게 된다.

$$\cos(Q, D_d) = \frac{Q \bullet D_d}{|Q||D_d|} = \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \bullet w_{d,t}$$

과 같이 표현할 수 있다. 여기서 문서 d의 길이 W_d, 즉 문서가중치는

$$W_d = \sqrt{\sum_{t=1}^{n} w_{d,t}{}^2}$$

질의의 가중치 W_q는

$$W_q = \sqrt{\sum_{t=1}^{n} w_{q,t}{}^2}$$

으로 계산된다. 여기에 TF*IDF 방식 ($w_{d,t} = f_{d,t} * \log(N/f_t)$) 을 적용하면

$$\cos(Q, D_d) = \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \bullet f_{d,t} \log \frac{N}{f_t}$$

가 된다. 실제 구현에서 $f_{q,t}$ 를 질의어에 존재하는 단어에 대해서는 1, 없으면 0으로 설정한다면 코사인 값은 다음과 같이 간단해질 수 있다.

$$\cos(Q, D_d) = \frac{1}{W_q W_d} \sum_{t \in Q} f_{d,t} (\log \frac{N}{f_t})^2 \cdots\cdots\cdots\cdots (1)$$

(1) 식에서 W_q는 주어진 질의에 대해서는 항상 일정한 값을 가지므로 순위 결정에서는 아무런 영향을 미치지 못한다. 표 1-1의 문서집합에 대해서 위의 식을 계산해 보자.

(a)

d	색인어 t					W_d
	a	B	C	d	e	
1	3	2	0	0	1	3.10
2	1	2	3	1	0	3.31
3	0	5	0	0	1	1.42
4	0	1	1	0	1	0.86
5	1	2	1	0	0	1.27
6	0	0	1	0	4	2.39
F_t w_t	3 1.00	5 0.26	4 0.58	1 2.58	4 0.58	

(b)

d	질의				
	D W_q = 2.58	C W_q = 0.58	c, d W_q = 2.64	a, b, e W_q = 1.18	a, b, c, d, e W_q = 2.90
1	0.00	0.00	0.00	0.95	0.39
2	0.78	0.53	0.88	0.29	0.92
3	0.00	0.00	0.00	0.40	0.16
4	0.00	0.67	0.15	0.40	0.30
5	0.00	0.46	0.10	0.76	0.40
6	0.00	0.24	0.05	0.48	0.24
최고 순위 문서	2	4	2	1	2

표 1-3. 코사인 가중치 방식의 적용: (a) 단어 빈도 $f_{d,t}$와 문서가중치 W_d; (b) 각 질의에 대한 코사인 가중치 계산 결과. 표 1-1에서 예로 보여준 문서 집합에 대해 코사인 측정법을 적용한 예이다.

벡터 공간 모델의 구현

식(1)을 대용량 문서 DB에 그대로 적용하려고 하면 계산을 하기위해 CPU에 많은 시간을 할당해야 하고 또한 많은 메모리가 필요하다. 어떻게 이러한 장애들을 극복해야 할 것인가? 여기에서는 코사인 측정법을 구현한다고 가정하고 각 과정을 설명해 보자.

문서내 단어빈도

f_t는 색인어 파일에 저장되어 있고, $f_{d,t}$는 포스팅 파일에서 계산할 수 있다. 또한 W_q는 각 질의에 대해 항상 일정한 값을 가진다. 따라서 식(1)에서 계산해야 될 항목은 W_d만이 남는다. 우선 W_d는 미리 계산되어 주메모리상에 올라와 있다고 가정하자. 색인어 별로 벡터공간모델을 실행하기위해서는 누산기(accumulator)라는 새로운 자료구조체가 하나 더 필요하다. 그림 1-1은 각 문서에 대해 $cosine(Q, D_d)$를 계산하고 최상위 r개의 문서를 가져오는 알고리즘을 보여주고 있다.

그림 1-1. 코사인 측정법을 이용한 상위 r개의 문서를 검색하는 알고리즘

Cosine measure를 이용하여 r개의 문서를 검색하기 위해서는

1. 누산기 A <- {} 설정. (A를 0으로 초기화)
2. 질의에 있는 용어 t 에 대하여
   a. t를 형태소 분석
   b. 색인파일에서 t를 찾아 해당하는 f_t를 기록하고 포스팅화일의 위치 I_t를 기록
   c. w_t <- $(\log(N/f_t))^2$로 설정
   d. I_t에 해당하는 포스팅화일의 데이터를 읽는다.
   e. 각 문서에 대해서 $f_{d,t}$를 계산하고

      누산기 A에 A_d가 없으면

      A_d <- 0

      A <- A + $\{A_d\}$

      A_d <- A_d + $f_{d,t} * w_t$
3. 누산기에 들어 있는 각 A_d에 대해서

   A_d <- A_d / W_d (W_q는 각 질의에 대해 고정된 값이므로, 이제 A_d는 $cosine(Q, D_d)$와 비례한다)
4. $1 <= i <= r$ 인 동안
   a. $A_d = \max\{A\}$인 d를 선택
   b. 문서 d를 가져온다.
   c. A <- A - $\{A_d\}$ 로 설정

위의 방법은 여러 가지 문제점을 안고 있다. 먼저 W_q(질의에 대해 일정)와 $f_{q,t}$(질의어가 있으면 1, 없으면 0로 설정)가 무시되었다는 것이고 둘째로는 대용량의 주메모리를 사용한다는 것이다. 여기에서 주목할 만한 점은 $r(r << N)$개의 문서만 가져올 것이므로 그림1-1의 제4단계에서 누산기 집합 A를 완전히 정렬할 필요는 없다는 것이다. 그러면, 문서가중치 W_d에 대한 메모리, 누산기 A에 대한 메모리, 누산기 A의 정렬에 대한 문제를 차례로 살펴보자.

문서가중치 W_d 저장 메모리에 대한 고찰

1백만 건의 문서가중치 W_d를 모두 주메모리상에 올리기 위해서는 4Mb(= sizeof(int) * 1,000,000 = 4byte * 1M = 4Mb)의 메모리가 필요하다. 그렇지 않고 저장시스템 내에 이 자료구조를 저장해둔다고 하더라도, 문서가중치를 계산하기 위해서는 문서를 적재할 때 과중한 시간이 더 소요될 것이다. 데이터베이스를 적재할 때 문서가중치를 계산하기 위해서는 각 문서에 대한 정보를 데이터베이스 적재가 끝나는 시점까지 유지하고 있어야 한다. 이 정보를 유지하는 시간과 메모리의 사용이 과다할 것으로 예상된다. Cosine 측정법을 사용하기 위해서는 미리 계산되어 있는 문서가중치가 필수적이지만 이 값을 계산하기위해 소요되는 노력(CPU, memory)과 시간이 이를 이용함으로써 높아지는 검색 결과의 효율성이 보상해줄 지는 일단 실험을 해보아야 할 것이다[1].

누산기 A 저장 메모리

색인 파일에서 질의어에 있는 모든 단어에 대해 f_t를 읽어 들이고 f_t가 증가하는 순서대로(w_t가 감소하는 순서대로) 포스팅 파일을 접근한다고 하면 더 이상 N개의 누산기가 필요하지는 않다. 여기에서 문제점은 순위결정이 더 이상 cosine 규칙을 따르지 않게 된다는 것인데 cosine 규칙 자체가 증험적 규칙이므로 별 문제가 되지는 않는다.

먼저 누산기의 크기를 고정시키고 그 이후에 나타나는 문서는 무시하는 것이다. 결과적으로 가중치가 높은 단어들이 누산기를 차지하게 된다. 누산기의 경계를 넘어가는 단어들에 대해서는 계속 포스팅 파일에서 데이터를 읽어 들여 누산기에 들어가 있는 문서들에 대해서는 cosine 값을 계속(**continue**) 계산한다. (continue 방식외의 다른 방식인 **quit** 방식은 누산기의 경계를 넘어가는 질의 단어들에 대해서는 코사인 값 계산을 하지 않는 데, **quit** 방식은 검색 속도는 빠르지만 검색 효율성이 많이 떨어진다고 한다.) *Continue* 방식은 누산기가 5,000개일 경우 최상위 1,000개의 문서를 효과적으로 검색했고, 누산기가 30,000개일 경우 검색 효율성이 코사인 방식보다 뛰어나다고 한다.

---

[1] 요즘 같이 대용량의 데이터베이스를 다루는 추세에서는 대용량 데이터베이스를 대상으로 작업을 해야 하므로 문서 적재시에 과중한 시간을 더 투자해야 할 것으로 예상된다. 따라서 KRISTAL-II가 cosine 측정법을 지원하게 되면 코사인 측정방식의 활용성에 비해 문서 적재의 효율성이 매우 떨어질 것으로 예상되며, 또한 on-line update/delete를 지원할 수 없다는 단점이 발생하므로 KRISTAL-II에서의 벡터공간모델 구현은 inner product 방식을 보완하는 쪽으로 방향을 잡았다.

그림 1-2. Pruning 알고리즘

질의에 들어있는 색인어들을 $IDF(w_t)$가 감소하는 순서대로 정렬 (f_t가 증가하는 순서대로)

1. 질의에 있는 색인어 t에 대하여
   a. $w_t > Max_{t \in Q}\{w_t\}/3$이면
      1. 색인어 t에 해당하는 포스팅화일의 항목을 읽어들인다.
      2. 각 레코드 ID에 해당하는 누산기가 없으면 누산기를 새로 할당하고 0으로 초기화한다.
      3. 각 레코드 ID에 해당하는 누산기에 $w_{d,t}$를 더한다.
   b. $w_t <= Max_{t \in Q}\{w_t\}/3$이면
      1. 색인어 t에 해당하는 포스팅화일의 항목을 읽어들인다.
      2. 각 레코드 ID에 해당하는 누산기가 있으면 $w_{d,t}$를 더한다.
      3. 없으면 *continue*
2. 누산기에 들어 있는 각 A_d에 대해서

   A_d <- A_d / W_d
3. $1 <= i <= r$ 인 동안
   a. $A_d = \max\{A\}$인 d를 선택
   b. 문서 d를 가져온다.
   c. A <- $A - \{A_d\}$ 로 설정

Pruning

Pruning은 벡터 공간 모델을 구현할 때 질의어를 포함하는 모든 문서에 대해 계산을 수행하는 것이 아니라 문헌 빈도가 낮은 일부 질의어에 대한 문서들에 대해서만 가중치를 계산하는 방식을 말한다. 그림 1-2는 Pruning을 통한 누산기 알고리즘을 보여주고 있다. 위의 방식에서는 질의어 가중치가 질의어 중에서 가장 큰 질의어 가중치의 1/3을 초과하는 문서에 대해서만 누산기에 할당하는 방식을 취하고 그 외의 문서에 대해서는 가중치 값만을 반영하고 있다[2].

[2] 현재 KRISTAL-II에서 구현한 벡터공간모델 질의모듈에서는 위의 pruning 방식과는 약간 다른 방식을 취하였다. 위의 방식에서는 질의어 가중치가 최대 질의어 가중치의 1/3이하이면 이후의 DocID에 대해서는 문서가중치만을 반영하는 방식이지만, KRISTAL-II에서는 누산기의 크기를 고정시킨 후 누산기가 채워질 때까지 DocID를 누산기에 채우고, 그 이후의 DocID에 대해서는 문서가중치만을 반영하는 방식을 취하였다.

2. KRISTAL-II에서의 Vector Space Model 구현

가중치 계산 방식

$$M(Q, D) = (\Sigma F_{TF} \bullet F_{IDF}^2)/F_{DL}$$

KRISTAL-II에서 벡터공간모델을 구현하기 위한 문서가중치 계산 공식을 문서내 색인어빈도(F_{TF}), 역문헌빈도(F_{IDF}), 문서길이(F_{DL})의 세가지 요소로 분리하여 각 요소에 대해 각기 다른 방식을 적용할 수 있도록 구현하였다. 각 요소를 정의하기 위해서는 method.h를 변경하면 된다. 각 요소에서 사용된 계산 방식들은 아래와 같다[3].

F_{TF} 색인어 빈도 요소

1. 원형

   $F_{TF} = tf$

2. Harman 1986

   $F_{TF} = log(1+tf)$

F_{IDF} 역문서 빈도 요소

1. 원형

   $F_{IDF} = log(N/df)$

2. Witten et al. 1994

   $F_{IDF} = log((N + 1)/df)$

3. Croft and Harper 1979

   $F_{IDF} = log((N-df)/df)$

---

[3] 각 요소에 대해서는 위에 나열된 방식 외에도 여러 가지 방법론이 있으나 KRISTAL-II의 저장 방식에서 지원하는 방식만을 사용하였다. 특히 KRISTAL-II는 문서단위가 아니라 섹션단위로 랭킹 정보를 저장하기 때문에 문서길이 요소의 3번과 4번은 일반적으로는 사용할 수 없다. 또한 Cosine 방식의 문서 길이 정보를 저장하고 있지 않기 때문에 Cosine 계산방식을 적용할 수 없었다.

4. Sparck Jones 1972

$$F_{IDF} = log(N/df) + 1$$

F_{DL} 문서길이 요소

1. 원형

$$F_{DL} = Wd$$

*Wd*는 문서에 나타나는 모든 색인어의 수.

2. 참고문헌 없음

$$F_{DL} = log(Wd)$$

3. 참고문헌 없음

$$F_{DL} = Ud$$

*Ud*는 문서에 나타나는 유일한 색인어들의 수

4. Harman 1986

$$F_{DL} = log(Ud)$$

가중치 계산 방식에 대한 실험

4,142건의 BLUE 데이터베이스에 대해 위의 방식들에 대한 간단한 실험을 수행하였다. 그 결과 색인어 빈도 요소는 2번, 역문헌 빈도 요소는 4번, 문서길이요소는 2번 방식과 4번 방식이 가장 뛰어난 성능향상을 보여주었다[4]. 이들의 조합을 첨자 M(method의 머리)과 각 방식으로 조합했을 때, M242와 M244가 가장 좋은 결과를 보여주었다는 말이 된다. 그 결과는 표 2-1과 같다.

문서길이요소 중 4번 방식은 다중 섹션을 검색할 때는 문제가 있고 실제로 검색 효율도 M242방식과 M244방식은 뚜렷한 성능의 차이를 보여주지 못했기 때문에 – 즉, 다중 섹션을 검색할 경우 문서길이는 먼저 섹션들의 길이를 합산하게 되면 두 섹션 이상에서 동일한 질의어가 출현할 경우에는 이중으로 합해지는 결과를 초래하기 때문에 “Unique Term Conunt”의 의미가 사라진다 -

---

[4] 이 실험은 4,142건의 BLUE 데이터베이스를 대상으로 수행했다. 질의어는 2개를 사용하였으며 적합문헌은 불리안 질의(OR)를 통해 검색된 결과 중에서 수작업으로 선정하였다. 그 결과 색인어 빈도 요소 2번, 역문헌빈도 요소 4번, 문서길이요소 2번을 사용하였을 경우 각 요소 1번, 1번, 1번을 사용했을 경우보다 질의 {자동차, 모델}, {정보, 검색}, {시스템, 관리}에 대해 각각 28%, 67%, 133%의 성능향상을 보여주었다. 그러나 이 실험은 데이터베이스가 소규모이고 질의가 2개의 단어로 매우 짧기 때문에 정확한 실험이라고 할 수 없다. 따라서 정확한 자료는 KRIST set을 대상으로 Recall과 Precision에 대한 실험을 다시 수행해야 한다.

실제 구현에서는 제외하여 M242 방식이 가장 좋은 효율을 보인다고 보고 다음과 같이 가중치 계산 방식을 일단 확정하였다.

$$M(Q, D) = (\Sigma \log(1+tf) \bullet (\log(N/df)+1)^2)/\log(Wd)$$

가중치 계산 방식의 정의는 method.h에서 바꿀 수 있도록 정의해 두었다. 현재는 위의 방식을 사용하고 있으나 앞으로 보다 많은 실험을 통해 여러 데이터베이스에서 가장 효율적으로 사용될 수 있는 가중치 계산 방식을 좀 더 연구해볼 필요가 있다. 이에 대해서는 KRIST Set으로 실험을 더 수행하려고 한다.

(a) Precision

질의	{자동차, 모델}	{정보, 검색}	{시스템, 관리}
M111	18/50=0.36	36/100=0.36	12/50=0.24
M242	23/50=0.46	60/100=0.60	28/50=0.56
성능향상(M242/M111)	1.28	1.67	2.33

(b) Recall

질의	{자동차, 모델}	{정보, 검색}	{시스템, 관리}
M111	18/23=0.78	36/71=0.51	12/33=0.36
M242	23/23=1.00	60/71=0.85	28/33=0.85
성능향상(M242/M111)	1.28	1.67	2.33

표 2-1. 가중치계산 방식에 따른 성능차이 비교. (a) Precision. 질의 {자동차, 모델}과 {시스템, 관리}는 질의결과 50개 중의 Precision(P_{50})이며, 질의 {정보, 검색}은 100개중의 Precision(P_{100})이다. (b) Recall. 질의 {자동차, 모델}과 {시스템, 관리}는 질의결과 50개 중의 Recall(R_{50})이며, 질의 {정보, 검색}은 100개중의 Recall(R_{100})이다. {자동차, 모델} 질의의 적합문헌은 23개, {정보, 검색} 질의의 적합문헌은 71개, {시스템, 관리} 질의의 적합문헌은 33개 문서였다.

벡터스페이스모델 사용법

```
int            SetNum, MemCnt; /*  질의 Set 번호 및 금번 질의의 결과 DocID 수 */
T_RESULT       DocIDList;
USERSECLIST    SecList;
USERTERMLIST   TermList;

/* …… DB 를 연다 …… */
while(프로그램 수행) {
     VECTOR_SecList_Init( &SecList );
     for (질의 대상 섹션 모두) {
          VECTOR_SecList_Insert( &SecList, 섹션이름, 섹션가중치 );
     }

     VECTOR_TermList_Init( &TermList );
     for (질의어 모두) {
          VECTOR_TermList_Insert( &SecList, 질의어, 질의어의 단어 가중치 );
     }

     FIRE_VectorRank( SecList, TermList, &SetNum, &MemCnt );

     DocIDList = (T_RESULT *) malloc( MemCnt * sizeof(T_RESULT) );
     FIRE_GetDocIDList( SetNum, DocIDList );
     /* DocIDList 로 부터 원하는 방식으로 결과를 출력한다. */
}
/* …… DB 를 닫는다 …… */
```

벡터 공간 모델의 전체적인 흐름

KRISTAL-II에서 벡터 공간 모델 질의는 다음과 같은 흐름으로 처리된다.

1. 사용자 입력(질의 정보, 섹션 정보)
2. 사용자 입력 질의 정보와 섹션 정보의 처리(내부구조로 변환)
3. 각 질의어를 포함하는 섹션별 DocID 집합 구해오기
4. 질의어별로 섹션들의 DocID를 재정렬
5. 문서빈도(tf)가 증가하는 순서대로 질의어를 재정렬
6. MAX_SORT_SIZE에 정의된 개수만큼의 DocID를 누산

7. 문서길이로 정규화하지 않은 상태에서 누산기를 문서 가중치에 의해 정렬
8. 상위 MAX_SET_SIZE만큼의 문서를 문서길이로 정규화
9. 상위 MAX_SET_SIZE만큼의 문서를 가중치에 의해 정렬
10. 사용자에게 결과 DocIDList를 반환

벡터 공간 모델을 위한 사용자 입력

KRISTAL-II는 다중 섹션을 지원하므로 벡터 공간 모델에서도 섹션을 선택하여 검색할 수 있도록 했다. 따라서 사용자는 질의뿐 아니라 섹션을 입력할 수 있으며, 섹션 및 질의어 각각에 대한 가중치를 부여할 수 있도록 했다. 섹션을 입력하지 않았을 경우에는 형태소 분석을 하는 모든 섹션에 대해 검색을 하도록 하였다[5].

사용자 입력용 섹션 정보 (USERSECLIST)

사용자는 USERSECLIST에 질의대상 섹션을 지정할 수 있다. 질의 대상으로 할 수 있는 최대 섹션 개수는 MAX_SEC_NUM으로 정의된다.

```
#define MAX_SEC_NUM   50      /* 질의에 사용가능한 최대 섹션수          */
#define MAX_TERM_NUM  100     /* 질의에 사용가능한 최대 질의어의 수     */
#define MAX_TERM_LEN  100     /* 섹션 및 질의어의 최대 길이             */

typedef struct
{
      char Name[MAX_TERM_LEN];     /* 섹션 이름                        */
      double Wgt;                  /* 섹션 가중치, default = 1.0       */
} USERSECINFO;

typedef struct {
      int           Cnt;
      USERSECINFO   Sec[MAX_SEC_NUM];
} USERSECLIST;
```

---

[5] 섹션을 많이 사용할수록 검색 성능이 저하된다. 따라서 일률적으로 여러 섹션에 대해 벡터 공간 모델을 적용하기를 원하는 경우에는 이 섹션들을 묶어서 COMBINED SECTION으로 합쳐서 검색하는 것이 좋다.

섹션 입력을 위한 사용자 API

1. VECTOR_SecList_Init(USERSECLIST SecList);

USERSECLIST SecList를 초기화(SecList.Cnt = 0)한다. 섹션 정보를 입력하지 않는 경우라도 이 함수로 섹션 리스트를 초기화시켜야 한다.

2. VECTOR_SecList_Insert(&SecList, char *섹션이름, double 섹션가중치);

SecList에 섹션이름과 섹션가중치를 추가한다. 그리고 섹션수를 1씩 증가시킨다. 섹션을 선택하지 않고 검색 가능한 모든 섹션에 대해서 검색하고자 할 때는 초기화 함수만 실행하면 된다. 이 경우 섹션의 가중치는 디폴트로 1.0으로 지정된다.

3. VECTOR_SecList_Display(&SecList);

현재까지 입력된 섹션들의 정보, 즉 섹션 이름과 섹션의 가중치를 출력한다.

사용자 입력용 질의어 정보 (USERTERMLIST)

사용자는 USERTERMLIST에 질의어를 입력할 수 있다. 각 질의에서 사용할 수 있는 최대 질의어의 개수는 MAX_TERM_NUM으로 정의된다.

```
typedef struct
{
      char term[MAX_TERM_LEN];     /* 질의어                          */
      double Wgt;                  /* 질의어 가중치, default = 1.0    */
} USERTERMINFO;

typedef struct {
      int           Cnt;                /* 질의어의 수             */
      USERTERMINFO  Term[MAX_TERM_NUM]; /* 질의어 정보             */
} USERTERMLIST;
```

질의어 입력을 위한 사용자 API

1. VECTOR_TermList_Init(USERTERMLIST TermList);

USERTERMLIST TermList를 초기화(TermList.Cnt = 0)한다.

2. VECTOR_TermList_Insert(&TermList, char *질의어, double 질의어가중치);

TermList에 질의어와 질의어가중치를 추가한다. 그리고 질의어 수를 1씩 증가시킨다.

3. VECTOR_TermList_Display(&TermList);

현재까지 입력된 질의어들의 정보, 즉 질의어와 해당 질의어의 가중치를 출력한다.

사용자가 입력한 섹션정보와 질의 정보의 처리

사용자가 입력한 섹션 정보는 시스템이 사용할 수 있도록 SECTIONINFO라는 구조체의 배열에 vs_FillSecInfo() 함수에 의해 저장된다. 사용자가 입력한 섹션정보는 SECTIONINFO 구조체 배열의 2번째 항목부터 저장하며, 첫번째 항목은 각 섹션에서 결과로 얻어온 DocID들의 정렬을 위해 시스템이 사용한다.

```
typedef struct
{
	char SecName[MAX_TERM_LEN];  /* 섹션이름                          */
	int IndexType;               /* 섹션의 색인 형태                  */
	double Wgt;                  /* 섹션 가중치                       */

	int DIDCnt;                  /* 섹션의 해당 DocID 갯수            */
	R_POSTINFO *RPostList;       /* DocID List                        */
} SECTIONINFO;
```

또 사용자가 입력한 질의 단어들은 vqTERMINFO라는 구조체의 배열에 vs_FillTermInfo() 함수에 의해 저장된다. 사용자 질의어 정보는 vqTERMINFO 구조체 배열의 2번째 항목부터 저장하며, 첫번째 항목은 정렬(누산기)을 위해 시스템이 사용한다.

```
typedef struct
{
	char term[MAX_TERM_LEN];     /* index term itself         */
	int IndexType;       /* Index type for this term          */
	int df;              /* document frequency                */
	T_SEARCHOP op;
	double idf;          /* Inverse Document Ferequency (IDF)       */
	double idf_2;        /* (IDF)^2                           */
	double Wgt;          /* Term Weight for Relevance feedback      */

	int SecDIDCntMax;    /* max size of RPostList retrieved each section   */
	int DIDCnt;          /* Total Size of DocIDList for this term   */
	R_POSTINFO *DocIDList;       /* DocID list containing this term         */
} vqTERMINFO; /* vector query term information */
```

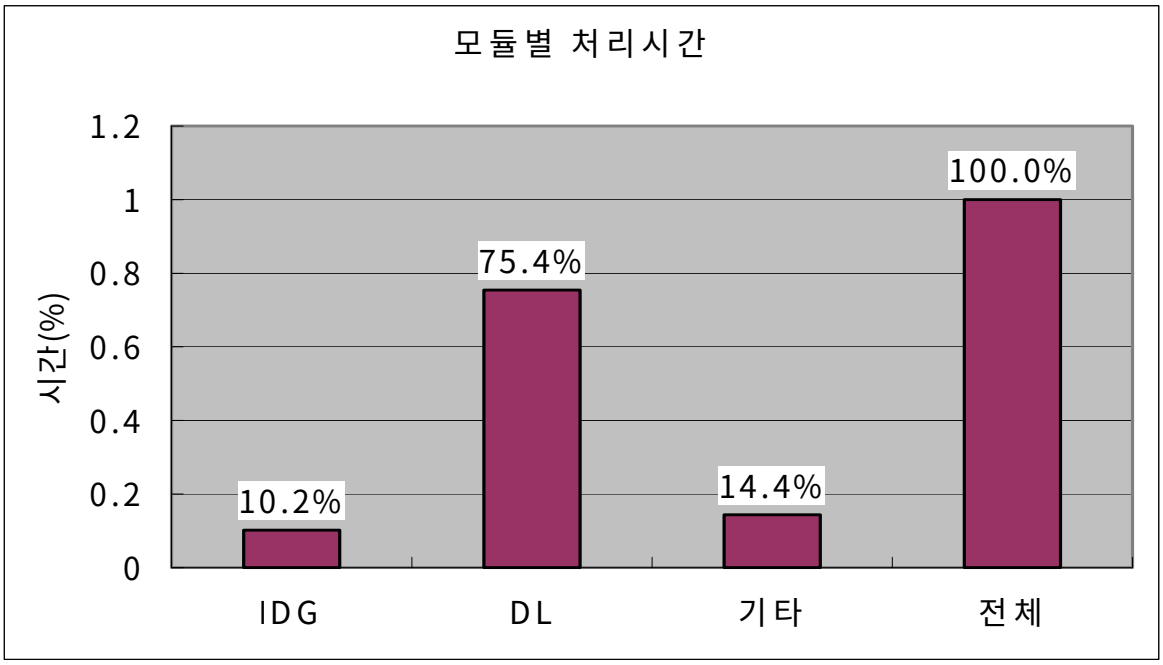


그림 2-1. 각 모듈별 질의 처리 시간 비교 (질의 처리 전체 시간에 대한 백분율). **IDG**: 각 질의어에 해당하는 DocID 목록을 가져오는 작업. **DL**: 문서길이로 정규화하는 작업. **기타**: 나머지 처리 시간. 누산기를 사용하지 않은 상태에서 질의를 처리할 경우 문서길이로 정규화하는 시간이 질의 처리의 가장 큰 병목 단계로 나타나는 것을 볼 수 있다.

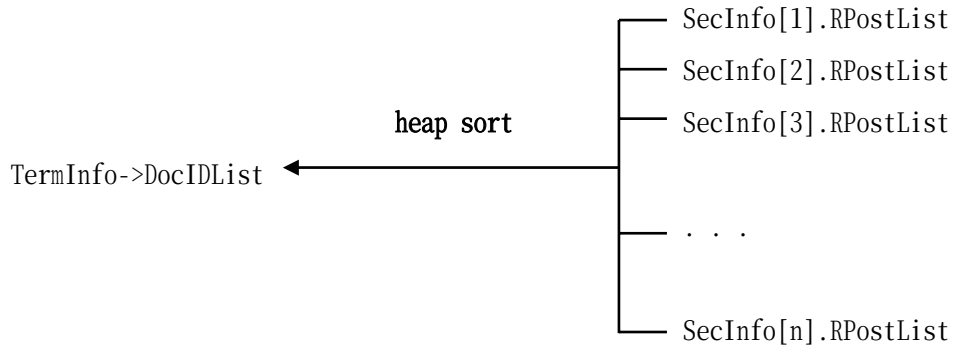
각 질의어에 대한 DocID 구하기 (vs_GetDocIDList() 함수)

각 질의어에 대해 먼저 SecInfo의 RPostList에 질의어가 나타나는 섹션들을 포함하는 DocID 목록을 가져온다.

```
while (질의어 i)
        SecInfo[i].RPostList <- 질의어 i 포함하는 DocID 목록
```

그런데 여기서 한가지 생각해야 될 문제가 있다. 현재는 모든 질의 단어에 대해 DocID를 통째로 가져오는 방식을 사용하고 있다. 질의내의 단어가 나타나는 문서가 많아질수록 DocID를 구해오는 시간의 비중이 매우 높아진다. 일반적으로 10%내외의 시간을 소모했으나 한국경제신문 데이터베이스에서 "한국", "경제", "기자" 등과 같이 질의어를 포함하는 문서의 수가 몇 만에 달할 경우에는 DocID를 구해오는 것이 전체 처리시간의 33 ~ 40%를 차지했으며, 심할 경우에는 60%에 달하기도 했다. 따라서 이 부분에 대한 질의 처리의 최적화가 요구된다. 앞으로 좀더 고려해야 할 부분이다.

각 섹션에 대해 DocID를 모두 구한 후에는 RPostList를 병합하여 해당 질의어의 TermInfo에 병합한다(vs_MergeSecDIDList_Heap() 함수). 병합하는 알고리즘은 heap sort를 이용하였다.



문서빈도(tf)에 의한 질의 정보의 정렬[6]

각 질의어를 가지는 DocID 정렬이 끝나면 질의 정보(vqTERMINFO 구조체)를 문서빈도가 증가하는 순서대로, 즉 IDF 값이 감소하는 순서대로, 정렬한다. 문서빈도가 작다는 것은 이 질의어가 데이터베이스 내에서 출현할 확률이 그만 작다는 것이고 따라서 보다 중요한 질의 정보라고 할 수 있다. 따라서 IDF 값에 의한 정렬은 다음 단계로 이루어질 이것은 누산을 하기 반드시 거쳐야 한다.

앞서도 언급했지만 질의어의 문서 빈도가 매우 높을 때는 이 질의어에 해당하는 DocID 를 구해오는 작업의 처리시간 비중이 매우 커지기 때문에 속도를 향상 시키기 위해서는 DocID 를 구해오기 전에 질의 정보를 먼저 df 증가하는 순서대로 정렬하고 문서빈도가 낮은 순서대로 누산을 하면서 DocID 를 구해오는 방식을 취해야 한다. 이 때 누산기가 차서 더 이상 새로운 DocID 를 구해올 필요가 없을 때는 나머지 질의어에 대해 현재 누산기에 포함되어 있는 DocID 중에서 나머지 질의어를 포함하는 DocID 들만에 대해서 문서정보를 가져와야 한다. 현재 이 방식은 구현되지않은 상태이며 앞으로 성능에 심각한 장애가 발생하면 고려해볼 문제이다.

각 질의어에 대한 DocIDList의 누산(Accumulation)

DocIDList의 누산은 초기에는 다음과 같은 2가지 방법으로 구현되었다. 첫번째 방법에서는 모든 질의어에 대해 DocID를 모두 구해서 누산한 후 문서길이로 정규화기 전에 MAX_SORT_SIZE에 정의된 개수만큼 문서 가중치에 의해 정렬한다(그림 2-2에서와 볼 수 있듯이 정렬 또한 처리할 양이 늘어나면 매우 많은 시간을 소비하므로 정렬할 자료의 양을 MAX_SORT_SIZE로 제한하였다). 그리고 상위 MAX_SET_SIZE 만큼의 문서에 대해 문서길이로 정규화시킨 후 이들만을 다시 문서가중치로

---

[6] 2개 이상의 복수 구조체들을 정렬하는 데는 heap sort를 이용하였으며(각 질의어당 섹션별로 DocIDList를 구하여 병합하는 부분과 누산기(accumulator)에서 사용), 그 외에 단일 구조체내에서 이루어지는 정렬에 대해서는 C 언어가 제공하는 qsort 라이브러리를 이용하였다(문서 가중치가 감소하는 순서대로 DocIDList를 정렬하는 작업). 그림 2-2에서 보이는 것처럼 quick sort에도 많은 처리시간이 소요되었다. 따라서 앞으로 속도를 향상시키기 위해서는 보다 빠른 정렬 알고리즘을 고려해야 할 듯 하다.

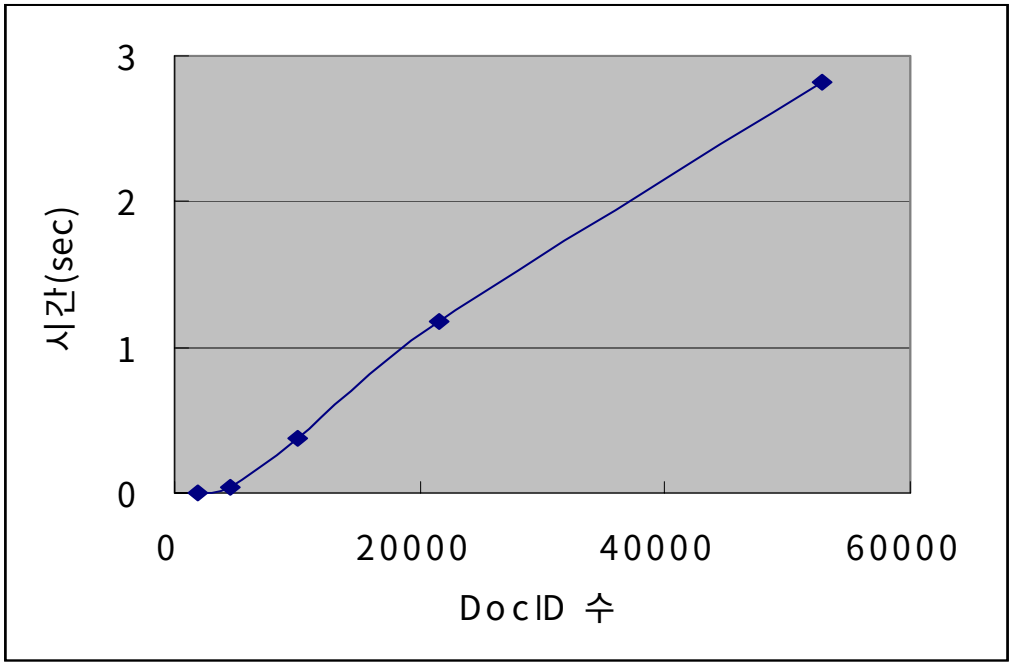


그림 2-2. 질의 처리에 있어서 가중치에 의한 qsort 정렬에 소요되는 시간 비교. Sun SparcCenter200E(메모리 1.2GB)에서 DocID 5000개일 경우 약 0.4초, 60000개일 경우 3초 정도가 소요되었으며 그래프에서와 같이 서로 정비례하는 것으로 나타났다. 정렬은 C 언어에서 제공하는 Quick Sort (qsort() 함수) 라이브러리를 이용하였다.

정규화시킨다. MAX_SORT_SIZE는 누산기의 크기를 의미하며 MAX_SET_SIZE는 사용자에게 넘겨줄 결과의 개수를 나타낸다. 두 번째 방법에서는 누산기의 크기까지만 DocID를 구해온다는 점에서 첫번째 방법과 다르며 나머지는 첫번째와 같은 방식을 거친다.

실험에 의하면 벡터공간모델에서의 병목은 문서의 길이(DocLen)을 구해오는 작업으로 전체 질의 처리 시간의 75.4%를 차지하였다(그림 2-1 참조). TF와 IDF를 이용하여 계산한 가중치는 문서길이로 정규화시켜야 하는데 KRISTAL-II에서는 이 정보를 UNIX 파일시스템에 저장하고 있으므로 이를 접근하는 데 많은 시간이 소요되는 것으로 추정된다. 따라서 이 부분에 대한 시스템 측면의 고려가 필요하고 이를 위하여 누산기의 크기를 제한할 필요가 있다. 그러나 문서길이 정규화 외의 처리시간은 질의 처리에 있어서 큰 비중을 차지하지 못하였기 때문에 두 번째 방법에서와 같이 처음부터 DocID의 개수를 제한할 필요는 없는 것으로 나타났으며 따라서 누산기의 크기를 제한하지 않는 첫번째 방식이 질의 결과 측면에서는 보다 효율적이라고 할 수 있다. 두번째 방법을 사용하여 한국경제신문 데이터베이스에 대해 "한국", "김대중", "김영삼" 등과 같이 문서 분포가 큰 질의어들만을 사용하면 질의 효율을 매우 떨어졌다. 그러나 첫번째 방식을 이용하면 주메모리를 많이

사용하기는 하지만[7] 처리시간은 거의 같으면서 질의 효율은 매우 좋아졌다.

첫번째 방식에 대해 좀더 고찰해보자. 첫번째 방식의 누산에서는 두번째 방식의 누산에 비해 정렬을 한번 더 사용하게 된다. 왜냐하면 첫번째 방식은 문서길이로 정규화하지 않은 MAX_SORT_SIZE만큼의 DocID들에 대해 가중치에 따라 1차적으로 정렬한 다음, 상위 MAX_SET_SIZE만큼의 DocID에 대해서만 문서길이로 정규화하고 이들만에 대해서 다시 정렬을 수행하기 때문이다. 이 방식은 색인어의 문서분포가 아주 큰 문서에 대해서는 매우 효율적으로 검색을 수행하는 것을 보여주었다. 그러나 문서 분포 빈도가 아주 큰 색인어와 문서분포 빈도가 아주 작은 색인어를 동시에 질의에 포함하면 심각한 검색 효율 저하를 초래하였다. 반면 두번째 방식을 이용하면 문서 빈도가 큰 색인어들만의 질의를 효율적으로 검색하지는 못했으나 문서빈도가 크기 때문에 적합한 문서들이 상위에 올라오게 되지만 이 문서들이 실제 최상위 문서라는 보장은 할 수 없었다.

이를 보완하기 위하여 첫번째 방식과 두번째 방식의 장점을 따서, 최종 방식에서는 두번째 방식에서 누산기 크기를 MAX_SORT_SIZE로 확장하였다. 이 최종 방식에서는 MAX_SORT_SIZE 만큼의 DocID가 누산되면 Pruning의 continue 방식처럼 이후의 DocID들에 대해서는 문서 가중치의 값만을 반영한다. 누산이 끝나면, 누산기를 가중치의 순서에 따라 정렬을 하게 되고 최상위 MAX_SET_SIZE만큼의 문서에 대해서 문서길이로 정규화하고, 다시 최상위 MAX_SET_SIZE 만큼의 DocID를 가중치에 따라 정렬을 한 후 사용자에게 반환하는 방식을 취하고 있다.

각 질의 처리 모듈의 처리시간 비교

지금까지 구현된 벡터공간모델은 34만 여 건의 문서 데이터베이스(원시문서의 크기는 약 450MB)를 대상으로 했을 때 문서빈도가 매우 큰 색인어들로 질의를 구성했을 때는 수행시간이 최대 7초에 달했다. 그러나 일반적인 질의에 대해서는 1 ~ 3초 (평균 2.58초) 정도의 질의 처리 시간이 소요되었다. 웹 환경에서는 그렇게 느린 편은 아니지만 서버/클라이언트 방식을 쓴다면 "제법 기다려야 한다"는 느낌을 주었다.

---

[7] 주 메모리 사용에 대해 고찰해 보자. 10만건의 결과를 처리해야 한다고 하면 일시적으로 사용되어야 할 메모리의 양은 DocID를 구해오는 구조체의 크기 12바이트x10만건 = 1.2MB이다. 그런데 첫번째 방식에서는 이 모두를 처리할 수 있도록 다시 메모리를 할당해야 하므로 순간적으로 이의 2배에 해당하는 2.4MB가 필요하다. 반면 두번째 방식에서는 MAX_SORT_SIZE에 정의된 DocID 구조체 수만큼 메모리를 다시 할당하므로, MAX_SORT_SIZE가 10,000으로 정의되어 있다고 하면 120kb가 더 요구될 뿐이다. 그러나 첫번째의 경우라 할지라도 이 정도의 메모리는 현재 하드웨어 상황을 고려할 때 PC 수준에서도 처리할 수 있으므로 문제가 되지 않을 것으로 보인다. 참고삼아, 한국경제신문 데이터베이스는 DB의 크기가 344,869건이었으며 가장 많이 나타나는 색인어는 '한국'으로 총 78,649개의 문서에서 색인되었다.

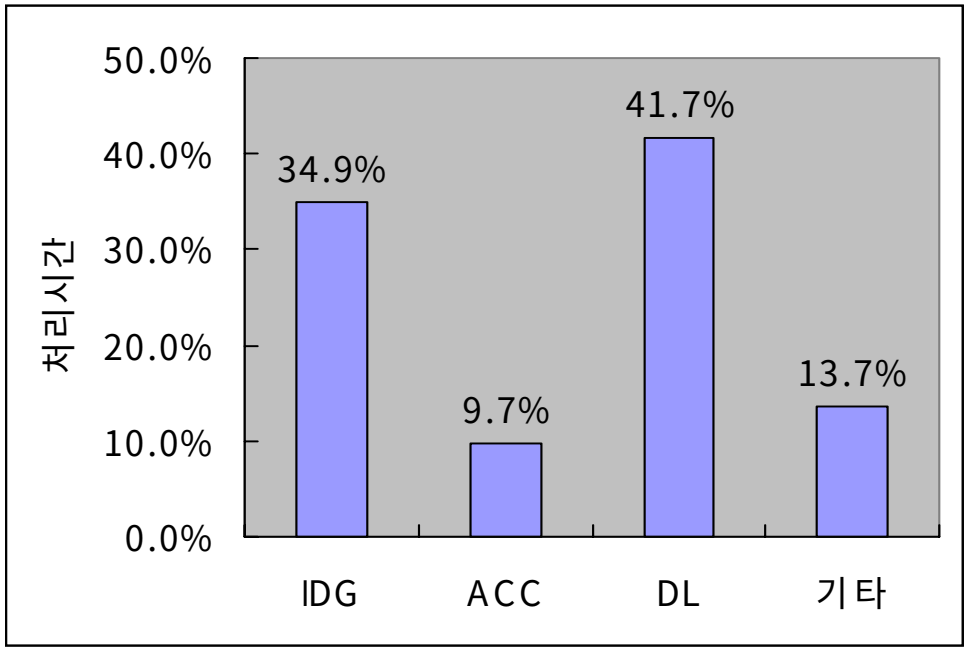
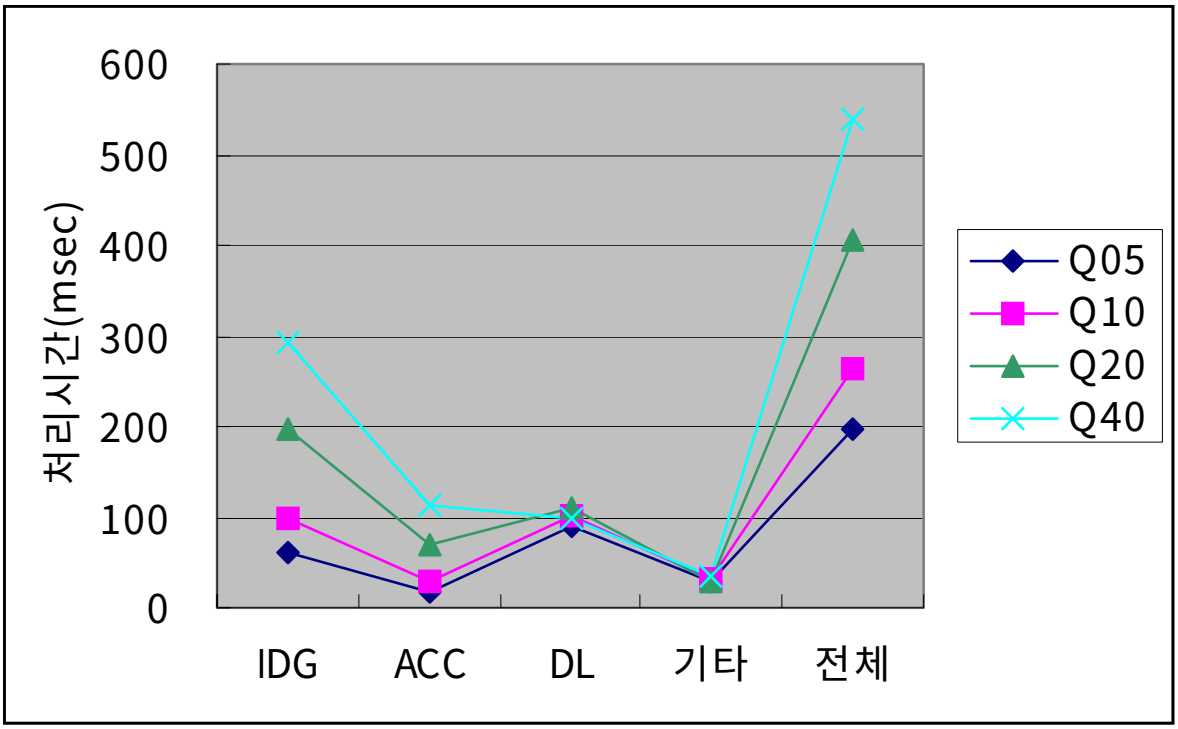


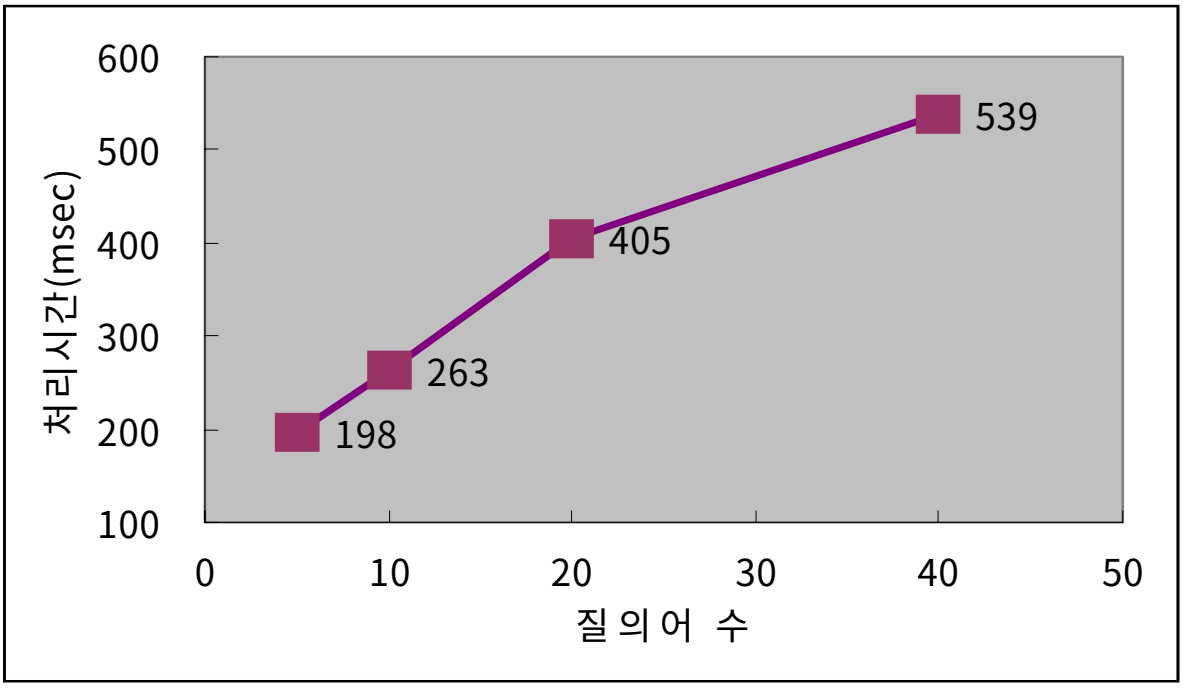
그림 2-3. KRISTAL-II 벡터 공간 모델의 각 처리 모듈별 처리시간 비교. **IDG**: 문서의 DocID를 구해오는 시간, **ACC**: 누산기 처리시간, **DL**: 문서길이로 정규화하는 시간, **기타**:.IDG, ACC, DL 처리시간을 제외한 나머지 시간. 20개의 질의에 대한 평균이며, 전체 질의 처리시간은 평균 2.58초였고, 각 모듈별로는 IDG가 0.90초, ACC가 0.25초, DL이 1.08초, 기타 0.35초가 소요되었다(본문참조).

그림 2-3은 한국경제신문 데이터베이스에서 질의 처리속도를 보여주고 있다. 앞서 언급한 바 있듯이 한국경제신문 데이터베이스는 총 344,869건의 문서로 이루어진 데이터베이스로 약 6년간의 기사를 담고 있다. 그림 2-3에서는 DocID를 구해오는 과정(DL), 누산기 모듈(ACC), 문서길이 정규화 모듈(DL)에 따라 전체 질의에서 차지하는 시간을 보여주고 있다. 이 그림은 5 ~ 20개의 질의어로 이루어진 20개의 질의를 하나의 섹션에 대해 검색하는 데 걸리는 시간의 평균을 보여주고 있다. 질의할 단어들은 무작위로 선택하였으며, 초기 누산기 크기(MAX_SORT_SIZE)는 5,000개로, 사용자에게 반환할 결과의 크기(MAX_SET_SIZE)는 500개인 조건에서, 20개의 질의에 대한 처리시간은 평균 2.58초가 소요되었다.

그림 2-1과 2-3을 비교해 보면 누산기를 사용함으로써 문서길이 정규화(DL) 시간의 비율이 약 70%에서 약 40%로 감소한 것을 볼 수 있다. 이 비율은 사용자에게 반환할 문서의 수를 줄임으로써 더 개선할 수도 있다(사용자에게 돌려주는 결과의 크기가 500개의 문서이면 과다하다고 예상되며, 알타비스타(http://altavista.digital.com)처럼 200개 정도의 문서로도 충분할 것이라고 예상된다.) 한가지 더 주의깊게 볼 것은 질의에 있는 각 단어를 포함하고 있는 DocID를 구해오는 작업(IDG)의 비중이 크게 늘어나는 것을 알 수 있다. 누산기를 사용하지 않는 상태에서는 IDG는 전체 시간의 약 10%을 차지했으나 누산기를 사용하는 상태에서는 약 35%까지 증가했다. 또한 경우에 따라서는 60%이상까지 증가하기도 했다.



(a)



(b)

그림 2-4. 질의어의 수에 따른 검색효율성 비교. Q05 – 단어 5개짜리 질의, Q10 – 단어 10개, Q20 – 단어 20개, Q40 – 단어 40개. (a) 각 모듈별 소요 시간 추이. IDG: DocID를 구해오는 작업, ACC: 누산기(accumulator), DL: 문서길이 정규화, 기타: IDG ACC DL을 제외한 나머지 시간, 전체: 질의 처리 시간 전체 (자세한 내용은 본문 참조) (b) 질의어 수에 따른 질의 처리 시간의 변화 추이. 질의 처리 시간과 질의어의 수는 비례하는 것을 볼 수 있다.

질의 길이와 처리시간에 대한 실험

그림 2-4와 표 2-2는 질의어의 수와 질의 처리시간에 대한 결과를 보여주고 있다. 무작위로 단어를 선정하여 질의어 5개(Q05), 10개(Q10), 20개(Q20), 40개(Q40) 짜리 질의를 각각 10개씩 만들어서 벡터공간모델 질의를 수행하였다. 그림 2-4(a) 및 표 2-2에서 보듯이 질의에 포함된 단어가 많아질수록 포스팅 파일에서 DocID를 구해오는 작업(IDG)의 비중이 계속 증가하는 것을 볼 수 있는데 이것은 질의가 길어질수록 처리해야 할 DocID의 수가 늘어나기 때문이라고 해석할 수 있다. 마찬가지로 누산기(ACC) 또한 질의가 길수록 처리 시간이 점차 증가하는 데 이는 힙을 이용한 DocID의 병합이 그 만큼 늘어나기 때문이다. 반면 문서길이 정규화(DL)는 MAX_SET_SIZE로 제한되어 있기 때문에 질의가 늘어나는 것에 상관없이 항상 일정한 속도를 유지한다고 볼 수 있다.

(단위: msec)

모듈 / 질의 set	IDG	ACC	DL	기타	전체
Q05	60	17	90	30	198
Q10	100	28	102	33	263
Q20	198	69	110	28	405
Q40	294	112	97	35	539
평균	163	57	100	32	351

표 2-2. 질의어의 수에 따른 각 모듈별 질의 처리시간 변화 추이. 평균은 각 모듈별 평균을 의미한다. 질의 set 당 10개씩 총 40개의 질의에 대해 평균 3.51초가 소요되었다.

결론 및 향후 계획

기본적인 벡터공간모델 모듈을 구현하여 KRISTAL-II에 통합하였다. 이 모듈은 질의어와 문서의 연관성을 계산하여 순위대로 MAX_SET_SIZE만큼 사용자에게 되돌려 준다. KRISTAL-II는 섹션단위를 지원하므로 벡터공간모델 모듈에서도 섹션을 지원하도록 하였으며, 섹션에 따라 가중치를 줄 수 있도록 하였다.

현재까지 개발된 벡터공간 모델 질의 모듈은 아직 완성형은 아니며 원형이라고 할 수 있다. 앞으로 보완해야 할 사항들은 다음과 같다.

- 검증된 test set 에서의 시험을 통한 가중치 계산 방식의 보완
- 각 질의어를 포함하는 DocID 를 구해오는 모듈의 효과적인 보완
- 보다 빠른 정렬 알고리즘을 이용한 성능향상
- 각 모듈별 처리 성능 향상(처리 시간의 측면에서)
- 기타 ……

여기에 보완하여 향후 2 가지 모듈이 더 필요하다. 첫째는 유사문서 찾기 기능(Find documents similar to this)이며 둘째는 질의 최적화(Query Refinement)이다.

유사문서 찾기

유사문서 찾기 기능은 일단 검색된 문서에 대해서 유사한 문서 찾기를 할 수 있도록 벡터공간모델을 확장하면 될 것이다. 이 모듈은 사용자의 질의 정보 입력을 문서 1 개로 대체하는 모듈로서 문서 내에서 출현하는 단어들의 정보를 취합하여 사용자 질의 정보 입력을 대체하도록 한다. 그러기 위해서는 형태소 분석 모듈을 사용해야 하며, 단어들의 가중치 계산 방식에 대한 연구가 더 필요하다. 우선 단일 문서에서 단어를 추출해내므로 문서길이 요소는 가중치 계산에서 제외할 수 있으므로 우선 다음과 같이 생각하기로 한다.

$$\text{단어 가중치} \quad W_t = TF * IDF^2$$

여기서 TF 와 IDF 는 벡터공간모델에서 사용한 수식과 동일한 것을 사용한다.

질의 최적화

유사문서 찾기 기능을 복수의 최상위 문서로 확장하면 질의 최적화를 할 수 있을 것이다. 그러나 적합성 피드백처럼 상위 문서내에 나타나는 단어들에 대한 문서간 분포와 이를 이용한 단어의 가중치 계산 방식을 고려해야 한다. 또한 위에서 살펴본 것처럼(그림 2-5), 질의어의 개수가 늘어날수록 질의 처리시간이 산술적으로 증가하기 때문에 몇 개의 단어에 대해서 질의 최적화를 할 것인지도 중요한 고려사항이 될 것이다. 이 부분은 향후에 적합성 피드백 과제가 완료되면 이 부분과 통합된 모듈을 구현하기로 한다.

3. 참고 문헌

Croft, W. B., and D. J. Harper. 1979. “Using Probabilistic Models of Document Retrieval Without Relevance Information.” *Documentation*, 35(4), 285-95

Harman, D. 1986. “An Experimental Study of Factors Important in Document Ranking.” Paper presented at ACM Conference on Research and Development in Information Retrieval, Cambridge, England.

Sparck Jones, K. 1972. “A Statistical Interpretation of Term Specificity and Its Application in Retrieval.” *J. Documentation*, 28(1), 11-20.

Witten, I. H., Moffat, A., and Bell, T. C. 1994. “Managing Gigabytes”, 141-148, Van Nostrand Reinhold, New York