Chapter 4.
Searching and Ranking

1. Search Engine?

- 크롤(crawl), 색인, 페이지 집합 검색하는데 필요한 단계들 소개
- 검색 결과 랭킹하는 방법

- 검색엔진 만드는 순서
  - 문서 수집 방법 개발
    - Crawling 또는 고정된 문서 컬렉션
  - 색인
    - 여러 단어들의 위치와 문서들을 담는 큰 테이블 생성하는 과정
  - 질의에 대해 랭킹된 문서 목록 제공

2. 단순 크롤러

- **크롤링(crawling)** or **스파이더링(spidering)**
  - 작은 페이지 집합에서 시작해서 페이지 내의 링크를 따라 다른 페이지들을 반복해서 찾는 과정

- Urllib2
  - 파이썬 번들 라이브러리
  - 웹 페이지들을 쉽게 다운로드 받을 수 있게

```
>>> from urllib2 import *
>>> c = urlopen('http://www.daegu.ac.kr')
>>> contents = c.read()
>>> len(contents)
139
>>> print contents[0:50]
<HTML>
<HEAD>


<meta http-equiv="Refresh" con
>>> contents
'<HTML>\r\n<HEAD>\r\n\r\n\r\n<meta http-equiv="Refresh" content="0;URL=http://ww
w.daegu.ac.kr/kor/index.asp">\r\n\r\n</HEAD>\r\n<BODY>\r\n</BODY>\r\n</HTML>\r\n
'

>>> c = urlopen('http://www.daegu.ac.kr/kor/index.asp')
>>> contents = c.read()
>>> print contents[0:200]
<!-- www1 -->
<html>
<head>
  <title>::: 교육 혁신 대학의 리더 - 대구대학교 :::</title>
  <link href="css/style.css" rel="stylesheet" type="text/css">
  <link rel="shortcut icon" href="/kor/favic
```

- 크롤러 코드
  - Beautiful Soup API 사용
    - 깨진 HTML로 만들어진 웹 페이지에서도 동작
  - Searchengine.py
    - Crawl 함수
      - 페이지 목록을 순회하면서 각 페이지마다 addtoindex 호출
      - Beautiful Soup 사용해서 페이지 내의 모든 링크 추출해 newpage 집합에 넣음
      - 과정 반복

3. 색인하기

- 색인
  - 단어 목록
    - 그 단어가 나타난 문서와 그 문서 안에서 나타난 위치 저장
  - 웹 페이지 내 텍스트만 대상
    - 구두점 등은 무시
- 색인 저장
  - 데이터베이스 이용
  - SQLite : 내장형 데이터베이스
- Crawler 클래스 작성

pysqlite

- SQLite 내장형 데이터베이스에 대한 파이썬 인터페이스
- SQLite
  - 별도의 서버 프로세스로 동작하지 않아 설치와 구성이 훨씬 쉬움
  - 전체 DB를 한 개의 파일에 저장
- 설치 및 사용법
  - 교재 : 부록 A (p.384)

  - 새로운 테이블 만들고 행 추가 후, 변경 사항 커밋
  - 테이블에 넣었던 내용 검색

```
>>> from pysqlite2 import dbapi2 as sqlite
>>> con = sqlite.connect('test1.db')
>>> con.execute('create table people (name, phone, city)')
<pysqlite2.dbapi2.Cursor object at 0x00CDD200>
>>> con.execute('insert into people values ("toby","111-2222","Daegu")')
<pysqlite2.dbapi2.Cursor object at 0x00CBFC50>
>>> con.commit()
>>> cur = con.execute('select * from people')
>>> cur.next()
(u'toby', u'111-2222', u'Daegu')
```

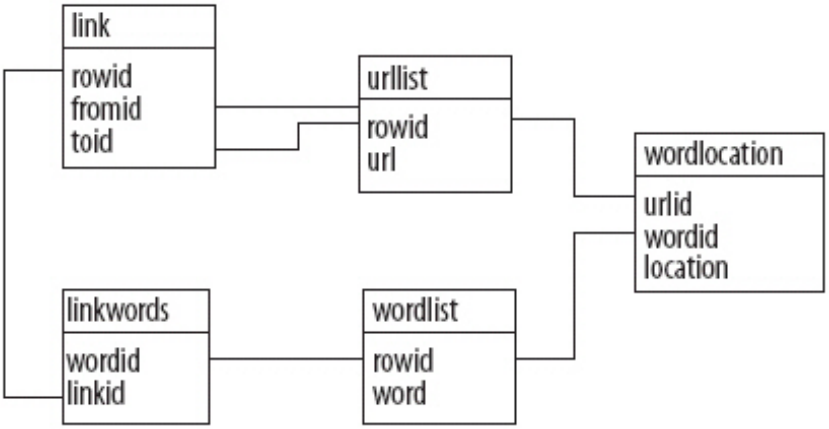
- Crawler 클래스

```
# Initialize the crawler with the name of database
def __init__(self,dbname):
  self.con=sqlite.connect(dbname)

def __del__(self):
  self.con.close()

def dbcommit(self):
  self.con.commit()
```

- 스키마 설정하기
  - 기본 색인용 스키마
    - 5개의 테이블로 구성
      - Urllist : 색인된 URL 목록
      - Wordlist : 단어 목록
      - Wordlocation : 문서 내의 단어 위치 목록
      - Link : 문서 간 링크 정보 저장
      - Linkwords : 링크에서 실제 사용된 단어 저장
  - 검색엔진 스키마



– Createindextables 함수

- 모든 테이블을 추가하는 함수
- 모든 테이블은 rowid라는 필드를 기본으로 가짐
- 검색 속도 향상을 위해 DB 색인 생성

```
# Create the database tables
def createindextables(self):
  self.con.execute('create table urllist(url)')
  self.con.execute('create table wordlist(word)')
  self.con.execute('create table wordlocation(urlid,wordid,location)')
  self.con.execute('create table link(fromid integer,toid integer)')
  self.con.execute('create table linkwords(wordid,linkid)')
  self.con.execute('create index wordidx on wordlist(word)')
  self.con.execute('create index urlidx on urllist(url)')
  self.con.execute('create index wordurlidx on wordlocation(wordid)')
  self.con.execute('create index urltoidx on link(toid)')
  self.con.execute('create index urlfromidx on link(fromid)')
  self.dbcommit()
```

```
>>> crawler = searchengine.crawler('searchindex.db')
>>> crawler.createindextables()
```

- 페이지 내 단어 찾기
  – 웹 페이지 내 텍스트 부분 추출
    - Soup 사용해서 텍스트 노드 찾고 내용 수집

    - Gettextonly 함수
      – 페이지 내의 모든 텍스트를 담고 있는 긴 문자열 리턴
      – 재귀함수
    - Separatewords 함수
      – 문자열을 단어 목록으로 분리
      – "알파벳이 아닌 문자"를 분리자로 사용
      – 완벽하지는 않음
        » 예) C++
        » 대안 : 스테밍 알고리즘

- 색인에 넣기
  - Addtoindex 함수
    - 페이지와 단어들을 색인에 추가
    - 문서 내 단어 위치와 연결 생성
  - Getentryid 함수
    - 항목의 ID 리턴
  - Isindexed 함수
    - 이미 DB에 등록된 웹 페이지인지 확인
    - 등록되었다면 그 안에 연관된 단어가 있는지 확인

```
>>> import searchengine
>>> page=['http://www.daegu.ac.kr']
>>> crawler = searchengine.crawler('searchindex1.db')
>>> crawler.crawl(page)
Indexing http://www.daegu.ac.kr
>>> page=['http://www.daegu.ac.kr/kor/index.asp']
>>> crawler.crawl(page)
Indexing http://www.daegu.ac.kr/kor/index.asp
Indexing http://woman.daegu.ac.kr/
Indexing http://web.daegu.ac.kr/dept/vsc/
Indexing http://enter.daegu.ac.kr/
Indexing http://club.cyworld.com/club/main/club_main.asp?club_id=52051156
```

  - 미리 색인된 DB
    - searchindex.db
    - 파이썬 코드가 있는 디렉토리에 복사

```
>>> cur=crawler.con.execute('select * from urllist')
>>> cur.next()
(u'http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.html',)
>>> cur.next()
(u'http://kiwitobes.com/wiki/Programming_language.html',)
>>> [row for row in crawler.con.execute(
        'select rowid from wordlocation where wordid=1')]
[(1,), (3739,), (6741,), (7464,), (7873,), (8336,), (9045,), (9577,), (10799,),
(11348,), (12095,), (12637,), (13306,), (15702,), (16325,), (18417,), (21440,),
```

4. 검색하기

- Searcher 클래스
  - 검색엔진의 검색부분
- 복합어 검색 필요
  - Getmatchrow 함수
    - 질의를 개별 단어로 분리한 후, 모든 단어를 담고 있는 URL 선택하도록 SQL 질의 생성
    - ID 10과 17 두 개의 단어로 된 검색어인 경우

```
select w0.urlid,w0.location,w1.location
from  wordlocation w0,wordlocation w1
where w0.urlid=w1.urlid
and   w0.wordid=10
and   w1.wordid=17
```

  - 복합어 검색 예시

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.getmatchrows('functional programming')
select w0.urlid,w0.location,w1.location from wordlocation w0,wordlocation w1 whe
re w0.wordid=144 and w0.urlid=w1.urlid and w1.wordid=19
```

5. 내용 기반 랭킹

- 검색어와 일치하는 페이지 추출
  - 리턴된 페이지들의 순서는 단순히 크롤된 순서
- 검색어와 페이지 내용을 기반으로 점수 계산
  - 단어 빈도
  - 문서 내 위치
    - 핵심 주제는 문서 도입부에 있을 가능성 높음
  - 단어 거리
    - 복합어 검색인 경우 문서 내에서도 근접거리에 출현해야

- Getscoredlist 함수
  - 검색된 결과를 평가점수와 함께 리턴
  - Weights=[] 으로 설정 시 실행 예시

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.query('functional programming')
0.000000        http://kiwitobes.com/wiki/XSLT.html
0.000000        http://kiwitobes.com/wiki/XQuery.html
0.000000        http://kiwitobes.com/wiki/Unified_Modeling_Language.html
0.000000        http://kiwitobes.com/wiki/SNOBOL.html
0.000000        http://kiwitobes.com/wiki/Procedural_programming.html
```

- 정규화 함수 (normalizescores 함수)
  - 입력 : URL ID, 점수로 구성된 사전
  - 출력 : 0 ~ 1 사이의 점수로 정규화

- 단어 빈도 (frequencyscore 함수)
  - Rows 안에 있는 모든 고유 URL ID를 항목으로 가지는 사전 만들고, 각 항목이 출현한 빈도수 저장
  - 큰 값이 좋도록 정규화

```
def frequencyscore(self,rows):
  counts=dict([(row[0],0) for row in rows])
  for row in rows: counts[row[0]]+=1
  return self.normalizescores(counts)
```

  - Getscoredlist 함수 안의 소스 수정
    - 검색 결과에 빈도점수 반영
      - weights=[(1.0, self.frequencyscore(rows))]

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.query('functional programming')
1.000000        http://kiwitobes.com/wiki/Functional_programming.html
0.262476        http://kiwitobes.com/wiki/Categorical_list_of_programming_
languages.html
0.062310        http://kiwitobes.com/wiki/Programming_language.html
0.043976        http://kiwitobes.com/wiki/Lisp_programming_language.html
```

- 문서 내 위치
  - 해당 페이지 안에서 검색 단어의 위치
  - 페이지가 검색에 적합한 경우
    - 페이지 상단 가까이나 제목줄에 나타나는 경우
  - Locationscore 함수

```
def locationscore(self,rows):
  locations=dict([(row[0],1000000) for row in rows])
  for row in rows:
    loc=sum(row[1:])
    if loc<locations[row[0]]: locations[row[0]]=loc

  return self.normalizescores(locations,smallIsBetter=1)
```

  - Getscoredlist 함수 안의 소스 수정
    - 검색 결과에 문서 내 위치 점수 반영
      - weights=[(1.0, self.locationscore(rows))]

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.query('functional programming')
1.000000        http://kiwitobes.com/wiki/Functional_programming.html
0.150183        http://kiwitobes.com/wiki/Haskell_programming_language.htm
l
0.149635        http://kiwitobes.com/wiki/Opal_programming_language.html
0.149091        http://kiwitobes.com/wiki/Miranda programming language.htm
```

- 하나의 지표가 모든 경우에서 뛰어날 수는 없다.
  - 여러 가중치 조합 사용해야
  - Getscoredlist 함수 안의 소스 수정
    - 검색 결과에 단어빈도 & 문서 내 위치 점수 반영
      - weights=[(1.5,self.locationscore(rows)),
        (1.0,self.frequencyscore(rows))]

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.query('functional programming')
2.500000        http://kiwitobes.com/wiki/Functional_programming.html
0.438190        http://kiwitobes.com/wiki/Categorical_list_of_programming_
languages.html
0.265997        http://kiwitobes.com/wiki/Lisp_programming_language.html
0.242920        http://kiwitobes.com/wiki/Haskell_programming_language.htm
```

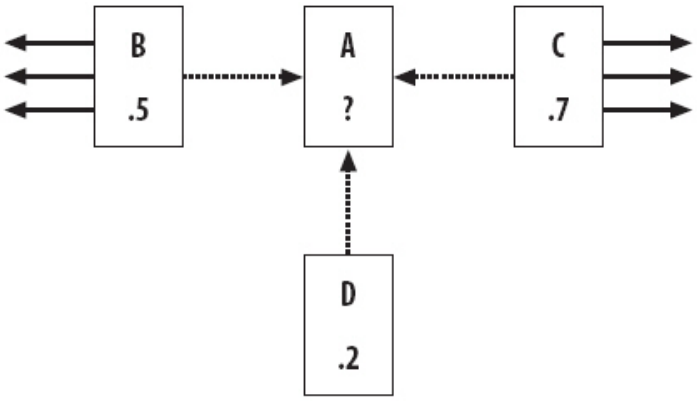
- 단어 거리
  - 검색어가 여러 단어인 경우, 검색 단어들이 페이지에서 서로 가까이 붙어 있는 결과 유용
  - 순서 바뀜과 검색 단어 사이에 있는 다른 단어들은 묵인
  - Distancescore 함수
    - Locationscore 함수와 유사
  - Getscoredlist 함수 안의 소스 수정
    - weights=[(1.0,self.locationscore(rows)),
      (1.0,self.frequencyscore(rows)),
      (1.0,self.distancescore(rows))]

```
>>> import searchengine
>>> e = searchengine.searcher('searchindex.db')
>>> e.query('functional programming')
3.000000        http://kiwitobes.com/wiki/Functional_programming.html
1.379619        http://kiwitobes.com/wiki/Categorical_list_of_programming_
languages.html
1.191990        http://kiwitobes.com/wiki/Lisp_programming_language.html
```

6. 유입 링크 사용하기

- 다른 사람들이 그 페이지에 제공한 정보를 적용하여 검색 품질 향상
- 단순 계산
  - 각 페이지의 유입 링크 수 계산
    - 페이지에 대한 지표로 사용
    - 인용 논문수로 논문의 중요도 평가
  - 지표 조작 가능
  - Inboundlinkscore 함수

- 페이지 랭크 알고리즘
  - 구글 창업자에 의해 개발됨
  - 인기 페이지로부터의 링크가 랭킹 계산에서 더 많은 가치 가지도록




```
PR(A) = 0.15 + 0.85 * ( PR(B)/links(B) + PR(C)/links(C) + PR(D)/links(D) )
      = 0.15 + 0.85 * ( 0.5/4 + 0.7/5 + 0.2/1 )
      = 0.15 + 0.85 * ( 0.125 + 0.14 + 0.2)
      = 0.15 + 0.85 * 0.465
      = 0.54525
```

- 페이지 A와 연결된 다른 페이지들의 페이지랭크값을 미리 알고 있어야
- 페이지랭크값을 초기에 임의의 값으로 두고, 여러 번 반복해서 계산
  - 반복할 때마다 실제 값에 점점 가까워짐

- 페이지랭크 계산
  - 시간이 많이 걸림
  - 검색어와 상관없기 때문에 미리 계산
  - 계산값을 테이블에 저장
  - Calculatepagerank 함수
    - Crawler 클래스에 추가

- PR 계산

```
>>> import searchengine
>>> e = searchengine.crawler('searchindex.db')
>>> e.calculatepagerank()
Iteration 0
Iteration 1
```

- 예제 DB에서 가장 높은 PR값 갖는 페이지 찾기

```
>>> cur = e.con.execute('select * from pagerank order by score desc')
>>> for i in range(3):
        print cur.next()


(438, 2.5285160000000002)
(2, 1.1614640000000001)
(543, 1.064252)
>>> s = searchengine.searcher('searchindex.db')
>>> s.geturlname(438)
u'http://kiwitobes.com/wiki/Main_Page.html'
```

- Pagerankscore 함수
  - Searcher 클래스에 추가
  - PR값 정규화

- Getscoredlist 함수 안의 소스 수정
  - weights=[(1.0, self.locationscore(rows)),
    (1.0, self.frequencyscore(rows)),
    (1.0, self.pagerankscore(rows))]

```
>>> import searchengine
>>> s = searchengine.searcher('searchindex.db')
>>> s.query('functional programming')
2.318146        http://kiwitobes.com/wiki/Functional_programming.html
1.074506        http://kiwitobes.com/wiki/Programming_language.html
0.517633        http://kiwitobes.com/wiki/Categorical_list_of_programming_
```

- 링크 텍스트 활용
  - 페이지에 대한 링크가 무엇을 말하는지 분석하면 유용한 정보
  - Linktextscore 함수
    - Searcher 클래스에 추가
  - Getscoredlist 함수 안의 소스 수정
    - weights=[(1.0, self.locationscore(rows)),
      (1.0, self.frequencyscore(rows)),
      (1.0, self.pagerankscore(rows)),
      (1.0, self.linktextscore(rows,wordids))]

- 모든 경우에 잘 동작하는 표준 가중치는 없음
  - 사용할 지표와 제공할 가중치는 구축하려는 응용에 따라 달라짐

Vector Model (1)

- 벡터 공간 검색 모델 (Vector Space Retrieval Model)
  - 메인 아이디어
    - 모든 것(문서, 질의, 단어)은 고차원 공간의 벡터이다.
  - SMART [Salton71]

- 문서 벡터 (Document Vector)
  - 각 단어가 하나의 차원에 해당
  - 각 문서는 하나의 벡터
    - $D_i = (t_{i1}, t_{i2}, \ldots, t_{in})$
    - $D_j = (t_{j1}, t_{j2}, \ldots, t_{jn})$

Vector Model (2)

- 문서 유사도의 측정
  - 어휘 유사도 : 의미 유사도의 추정
  - 예 : 두 벡터 사이의 코사인(cosine) 값

$$Similarity\ (D_i, D_j) = \frac{D_i \bullet D_j}{\|D_i\| \times \|D_j\|} = \frac{\sum_{k=1}^{n} t_{ik} \times t_{jk}}{\sqrt{\sum_{k=1}^{n} t_{ik}^{\ 2}} \times \sqrt{\sum_{k=1}^{n} t_{jk}^{\ 2}}}$$

Vector Space Retrieval (1)

- 예
  - 문서의 벡터 공간이 다음의 세 단어에 의해 정의된다고 가정
    - hardware, software, users
  - 이에 따른 문서 집합은 다음과 같이 표현된다 가정

    D1 = (1, 0, 0), D2 = (0, 1, 0), D3 = (0, 0, 1)

    D4 = (1, 1, 0), D5 = (1, 0, 1), D6 = (0, 1, 1)

    D7 = (1, 1, 1), D8 = (1, 0, 1), D9 = (0, 1, 1)
  - 질의 : "hardware and software"
    - 질의 벡터 : Q = (1, 1, 0)
  - 어떤 문서가 검색되어야 하는가?

Vector Space Retrieval (2)

- 벡터 질의의 매칭
  - 질의-문서간 유사도 계산

    S(Q, D1) = 0.71 S(Q, D2) = 0.71 S(Q, D3) = 0

    S(Q, D4) = 1 S(Q, D5) = 0.5 S(Q, D6) = 0.5

    S(Q, D7) = 0.82 S(Q, D8) = 0.5 S(Q, D9) = 0.5

    (참고) sqrt(2) = 1.414, sqrt(3) = 1.732, sqrt(6) = 2.449
  - 검색된 문서 집합 (순서대로)
    - {D4, D7, D1, D2, D5, D6, D8, D9}
- 불린 질의의 매칭이었다면?
  - AND : D4, D7 검색
  - OR : D1, D2, D4, D5, D6, D7, D8, D9 검색

가중치를 가지는 벡터 모델

- 단어 가중치
  - 단어의 존재 유무(0/1)가 아니라 단어의 중요도에 따른 수치 표현

    D1 = (0.7, 0.5, 0.3)

    D2 = (0.5, 0.2, 0.7)

    D3 = (0.3, 0.6, 0.9)

    D4 = (0.7, 0.9, 1.0)

- 질의도 역시 가중치를 줄 수 있다.

  Q = (0.7, 0.3, 0)

단어에 가중치 부여 하기

- 표준적인 가이드라인은 없다.
- 일반적인 선택
  - 이진
    - 문서에 단어가 있으면 1, 없으면 0
  - **tf x idf**
    - 문서 내에서의 특성
      - f_{ik} : 문서 i에서 단어 k가 나타난 빈도수 (tf : term frequency)
      - 문서길이에 따른 정규화 필요
        - » 예 : **f_{ik} / 문서길이**
    - 문서 간의 특성
      - d_k : 단어 k를 포함하는 문서의 수
      - N : 문서의 수
      - **$\log_2(N / d_k)$** : 역 문서 빈도수 (idf : inverse document frequency)
        - » 단어 k가 도움되는, 즉 변별력있는 단어인지 평가

문서 A의 단어 가중치 계산의 예

문서 **A**의 통계정보

단어	빈도수
hardware	**4**
software	**4**
user	**4**
문서길이	**30**

문서 **B**의 통계정보

단어	빈도수
hardware	**5**
system	**10**
user	**3**
문서길이	**40**

문서 **C**의 통계정보

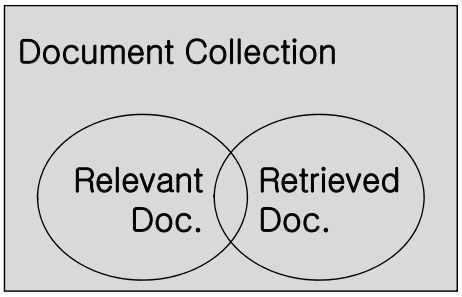
단어	빈도수
printer	**2**
disk	**4**
user	**1**
문서길이	**20**

- 단어 "**hardware**"의 가중치
  tf = 4 / 30 = 0.133
  idf = $\log_2(3 / 2) = 0.585$
  tf x idf = 0.078

- 단어 "**software**"의 가중치
  tf = 4 / 30 = 0.133
  idf = $\log_2(3 / 1) = 1.585$
  tf x idf = 0.211

- 단어 "**user**"의 가중치
  tf = 4 / 30 = 0.133
  idf = $\log_2(3 / 3) = 0$
  tf x idf = 0

Precision & Recall (1/2)



	Retrieved	Not retrieved
Relevant	W	X
Not relevant	Y	Z

<Contingency Table>

$$\text{정확률}: \quad P = \frac{\text{검색된 문서중 관련된 문서의 수}}{\text{검색된 문서의 총 수}} = \frac{W}{W+Y}$$

$$\text{재현율}: \quad R = \frac{\text{검색된 문서중 관련된 문서의 수}}{\text{관련된 문서의 총 수}} = \frac{W}{W+X}$$

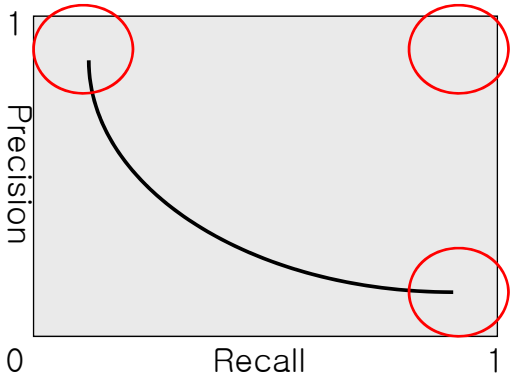
Precision & Recall (2/2)

- 정확률
  - 검색이 얼마나 정확한가
  - 정확률이 높을수록 관련 없는 문서의 검출이 적어진다.
- 재현율
  - 검색이 얼마나 완벽한가
  - 재현율이 높을수록 검색에서 빠지는 관련 문서의 수가 적어진다.
- 앞의 P/R 정의에서
  - 두 개의 수만 알 수 있다.
    - W+Y : 검색된 문서의 총 수
    - W : 검색된 문서 중 관련된 문서의 수
  - 관련된 문서의 전체 수(W+X)는 일반적으로 알 수 없다.

P와 R의 관계

- 이론적으로
  - P와 R은 상호 독립적이다.
- 실제적으로
  - 정확률의 희생을 통해 재현율이 증가
  - 재현율의 희생을 통해 정확률이 증가

대부분 정답문서만 검출하나 빠진 정답도 많이 존재



The ideal (P=1, R=1)

대부분의 정답문서 검출하나 쓸모없는 문서도 다수 검출

예) 정확률과 재현율의 계산

N	Doc #	Relevant	Recall	Precision
1	35	O	1 / 5 = 0.2	1 / 1 = 1.0
2	56	O	2 / 5 = 0.4	2 / 2 = 1.0
3	212		2 / 5 = 0.4	2 / 3 = 0.67
4	2	O	3 / 5 = 0.6	3 / 4 = 0.76
5	49		0.6	3 / 5 = 0.60
6	312	O	4 / 5 = 0.8	4 / 6 = 0.67
7	27		0.8	4 / 7 = 0.57
8	16		0.8	4 / 8 = 0.50
9	8		0.8	4 / 9 = 0.44
10	173		0.8	4 / 10 = 0.40
11	512		0.8	4 / 11 = 0.36
12	65		0.8	4 / 12 = 0.33
13	13	O	5 / 5 = 1.0	5 / 13 = 0.38
14	79		1.0	5 / 14 = 0.36

Single-Valued Measures

- E-Measure (by van Rijsbergen)
  - 정확률과 재현율의 상대적 중요성을 사용자가 명시
    - 사용자 정의 파라미터 : $\alpha = 1/(\beta^2+1)$, $0 \le \alpha \le 1$
    - α가 커질수록 정확률의 중요성 증가
  - 특수한 경우
    - $\alpha = ½$ ($\beta = 1$) : 동등한 중요성
    - $\alpha \to 0$ ($\beta \to \infty$) : 정확률 무시
    - $\alpha \to 1$ ($\beta \to 0$) : 재현율 무시
  - **F-measure** = 1 – E

$$E = 1 - \frac{1}{\alpha(1/P) + (1-\alpha)(1/R)} \qquad F = \frac{(\beta^2+1)PR}{\beta P + R}$$

- E 척도는 낮을 수록, F 척도는 높을 수록 좋은 성능
- 시스템별 절대적인 비교는 힘들다
  - 보는 측면에 따라 다르게 평가해야