Capítulo 4
Búsqueda y clasificación

Este capítulo trata sobre los motores de texto completo, los cuales permiten buscar una lista de palabras en un gran conjunto de documentos, y que clasifican los resultados de acuerdo a cuán relevantes son estos documentos para nosotros. Los algoritmos para las búsquedas de texto completo son los algoritmos más importantes de inteligencia colectiva, y muchas fortunas se han labrado por nuevas ideas en este campo. Es ampliamente conocido el rápido ascenso de Google de un proyecto académico al motor de búsqueda más popular del mundo basado en gran parte en el algoritmo PageRank, una variante de este es lo que aprenderás en este capítulo.

La recuperación de información es un gran campo con una larga historia. Este capítulo solamente puede abarcar algunos pocos conceptos clave, pero lo haremos mediante la construcción de un motor de búsqueda que indexará un conjunto de documentos y te dejará con ideas de cómo mejorarlas en el futuro. Por tanto el enfoque estará en los algoritmos para búsqueda y clasificación en lugar de los requerimientos de infraestructura para indexar grandes porciones de la Web, el motor de búsqueda que construirás no debe tener problemas con colecciones de hasta 100,000 páginas. A lo largo de este capítulo, aprenderás los pasos necesarios para rastrear, indexar y buscar un conjunto de páginas, e incluso clasificar los resultados de muchas formas distintas.

¿Qué es un motor de búsqueda?

El primer paso para crear un motor de búsqueda es desarrollar una forma para recolectar los documentos. En algunos casos, esto puede involucrar crawling (empezar con un pequeño conjunto de documentos y seguir los links a otros) en otros casos se puede empezar con una colección fija de documentos, por ejemplo en una intranet corporativa.

Después de que hayas recolectado los documentos, estos necesitan ser indexados. Esto involucra usualmente crear una gran tabla de los documentos y sus localizaciones para todas las palabras diferentes. Dependiendo de la aplicación particular, los documentos mismos no necesitan ser almacenados en una base de datos; el índice simplemente tiene que almacenar una referencia (tal como un path o una URL) de sus ubicaciones.

El paso final es, por supuesto, retornar una lista clasificada de los documentos a partir de una consulta. Recuperar cada documento con un conjunto dado de palabras es bastante sencillo una vez que tienes un índice, pero la magia real de esto es cómo los resultados son ordenados. Una gran cantidad de métricas se pueden generar, y son abundantes las formas en que se pueden ajustar para cambiar el orden de

clasificación. El aprender las diferentes métricas podría hacer que desees que los grandes motores de búsqueda te permitan un mayor control de ellos (“¿Por qué no le puede decir a google que mis palabras deben estar juntas?. Este capítulo analizará varias métricas basadas en el contenido de la página, tales como la frecuencia de las palabras, y luego abarca métricas basadas en información externa al contenido de la página, tales como el algoritmo Page-Rank, el cual considera cómo otras páginas enlazan la página en cuestión.

Finalmente, construirás una red neuronal para consultas de rankings. La red neuronal aprenderá a asociar búsquedas con resultados basados en qué links la gente clic después que ellos obtienen una lista de resultados de búsqueda. La red neuronal usará esta información para cambiar el orden de los resultados para mejorar reflejando lo que la gente ha clicado en el pasado.

Para trabajar con los ejemplos de este capítulo, necesitarás crear un módulo de Python llamado *searchengine*, el cual tiene dos clases: una para hacer crawling y creación de la base de datos, y la otra para hacer búsquedas de texto completo mediante consultas a la base de datos. Los ejemplos utilizaran SQLite, pero se pueden adaptar fácilmente para trabajar con un cliente de base de datos tradicional.

Para empezar, crear un nuevo archivo llamado searchengine.py y agregar la clase crawler y las declaraciones de sus métodos, la cual completaremos a lo largo de este capítulo.

```
class crawler:
  # Inicializa el crawler con el nombre de la base de datos
  def __init__(self,dbname):
    pass
  def __del__(self):
    pass
  def dbcommit(self):
    pass
  # Función auxiliar para obtener un ID de entrada y adiciona
  # este si no está presente
  def getentryid(self,table,field,value,createnew=True):
    return None
  # Indexa una página individual
  def addtoindex(self,url,soup):
    print 'Indexing %s' % url
  # Extrae el texto a partir de una pagina HTML (sin etiquetas)
  def gettextonly(self,soup):
    return None
  # Separa las palabras por un caracter que no sea espacio en blanco
  def separatewords(self,text):
    return None
  # Devuelve True si este URL ya esta indexado
  def isindexed(self,url):
    return False
  # Agrega un link entre dos paginas
```

```
  def addlinkref(self,urlFrom,urlTo,linkText):
    pass
  # Iniciando con una lista de páginas, hace una busqueda
  # primero en anchura hasta la profundidad dada
  # indexando las paginas que lleguen
  def crawl(self,pages,depth=2):
    pass
  # Crea las tablas de la base de datos
  def createindextables(self):
    pass
```

Un Crawler simple

Asumiré por ahora que no tienes una gran colección de documentos HTML residiendo en tu disco duro esperando ser indexados, así que voy a mostrarte como construir un *crawler* simple. Este será iniciado con un pequeño conjuntos de páginas a indexar y entonces seguiremos cualquier link en esta página para encontrar otra página, cuyos links también seguiremos. Este proceso es llamado *crawling* o *spidering*.

Para hacer esto, tu código tiene que descargar las páginas, pasarlas al indexador (el cual construirás en la siguiente sección), y entonces haremos el análisis léxico de las páginas para encontrar todos los links de las páginas que serán rastreadas luego. Afortunadamente, tenemos un par de bibliotecas que pueden ayudar con este proceso.

Usando urllib2

Urllb2 es una librería incluida con Python que hace fácil descargar páginas, todo lo que tienes que hacer es proporcionar la URL. La usarás en esta sección para descargar las páginas que serán indexadas. Para verla en acción inicia tu intérprete de Python y prueba esto:

```
>> import urllib2
>>
c=urllib2.urlopen('https://en.wikipedia.org/wiki/Programming_language')
>> contents=c.read( )
>> print contents[0:50]
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Trans'
```

Todo lo que tienes que hacer para almacenar el código de la página HTML es crear una conexión y leer su contenido.

Código del Crawler

El *crawler* usará el API BeautifulSoup, una biblioteca excelente que construye una representación estructurada de las páginas web. Es bastante tolerante con las páginas

web con HTML roto, lo cual es útil cuando construimos un *crawler* porque nunca sabes qué páginas podrías encontrarte.

Usando urllib2 y BeautifulSoup se puede construir un *crawler* que tome una lista de URLs para indexar y rastrear sus links para encontrar otras páginas para indexar. Primero, agregue estas sentencias `import` en la cabecera de *searcengine.py*:

```
import urllib2
from BeautifulSoup import *
from urlparse import urljoin
# Crea una lista de palabras a ignorar
ignorewords=set(['the','of','to','and','a','in','is','it'])
```

Ahora puedes rellenar el código para la función rastreadora. No guardaré ninguno de los rastreos aún, pero se imprimirán las URLs para que se pueda ver cómo trabaja. Necesitas poner esto al final del archivo (por lo que es parte de la clase `crawler`)

```
  def crawl(self,pages,depth=2):
    for i in range(depth):
      newpages=set( )
      for page in pages:
        try:
          c=urllib2.urlopen(page)
        except:
          print "Could not open %s" % page
          continue
        soup=BeautifulSoup(c.read( ))
        self.addtoindex(page,soup)
        links=soup('a')
        for link in links:
          if ('href' in dict(link.attrs)):
            url=urljoin(page,link['href'])
            if url.find("'")!=-1: continue
            url=url.split('#')[0]  # quitar la porcion de localizacion
            if url[0:4]=='http' and not self.isindexed(url):
              newpages.add(url)
            linkText=self.gettextonly(link)
            self.addlinkref(page,url,linkText)
        self.dbcommit( )
      pages=newpages
```

Esta función itera sobre la lista de páginas, llamando `addtoindex` para cada una (por ahora no hace nada más que imprimir la URL, pero rellenarás esto en la siguiente sección). Luego utiliza *Beautiful Soup* para obtener todos los links en una página y agregar sus URLs a un conjunto llamado `newpages`. Y al final de las iteraciones, `newpages` se convierte en `pages`, y el proceso se repite.

Esta función puede ser definida recursivamente de forma que cada link llama la función nuevamente, pero hacer una búsqueda en amplitud que permita la fácil modificación del código después, ya sea para mantener el rastreo continuamente o

para salvar una lista de páginas no indexadas para rastrearlas después. Esto también evita el riesgo de desbordamiento de la pila.

Puedes probar esta función en el intérprete de Python (no necesitas dejarlo terminar, así que presiona CTRL-C cuando estés aburrido):

```
>> import searchengine
>> pagelist=['http://es.wikipedia.org/wiki/Perl']
>> crawler=searchengine.crawler('')
>> crawler.crawl(pagelist)
Indexing http://kiwitobes.com/wiki/Perl.html
Could not open http://kiwitobes.com/wiki/Module_%28programming%29.html
Indexing http://kiwitobes.com/wiki/Open_Directory_Project.html
Indexing http://kiwitobes.com/wiki/Common_Gateway_Interface.html
```

Notarás que algunas páginas están repetidas. Hay un marcador en el código para otra función, `isindexed`, la cual determina si una página ha sido indexada recientemente antes de agregarla a `newpages`. Esto te permitirá correr esta función en cualquier lista de URLs en cualquier momento sin preocuparte de hacer trabajo innecesario.

Construyendo el índice

El siguiente paso es configurar la base de datos para el índice te texto completo. Como mencioné anteriormente, el índice es una lista de todas las palabras diferentes, junto con los documentos en los cuales ellas aparecen y sus ubicaciones en el documento. En este ejemplo, estarás viendo en el texto actual de la página e ignorando elementos no textuales. También estarás indexando palabras individuales con todos los caracteres de puntuación removidos. El método para separar palabras no es perfecto, pero es suficiente para construir un motor de búsqueda básico.

Debido a que abarcar diferente software de base de datos o configurar un servidor de base de datos está fuera del alcance de este libro, este capítulo te mostrará cómo almacenar el índice usando SQLite. SQLite es una base de datos embebida que es muy fácil de configurar y almacena una base de datos entera en un único archivo. SQLite utiliza SQL para las consultas, de modo que no debe ser difícil convertir el código de ejemplo para usar diferentes bases de datos. A partir de la versión 2.7 de Python forma parte de la biblioteca estándar como `sqlite3`.

Una vez que tengas instalado SQLite, agrega esta línea al principio de *searchengine.py*:

```
import sqlite3 as sqlite
```

También necesitarás cambiar los métodos `__init__`, `__del__`, y `dbcommit` para abrir y cerrar la base de datos:

```
  # Inicializa el crawler con el nombre de la base de datos
  def __init__(self,dbname):
    self.con=sqlite.connect(dbname)
```

```
def __del__(self,dbname):
  self.con.close()

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

Configuración del esquema

No corras el código justo aquí – aún necesitas preparar la base de datos. El esquema para la inexación básica es de cinco tablas. Las primera tabla (`urllist`) es la lista de URLs que han sido indexados. La segunda tabla (wordlist) es la lista de las palabras, y la tercera tabla (wordlocation) es una lista de las ubicaciones de las palabras en los documentos. Las restantes dos tablas especifican enlaces entre documentos. La tabla `link` almacena dos IDs de URL, indicando un enlace desde una tabla a otra, y `linkwords` usa las columnas `wordid` y `linkid` para almacenar qué palabras se usan actualmente y en que enlace. El esquema se muestra en la

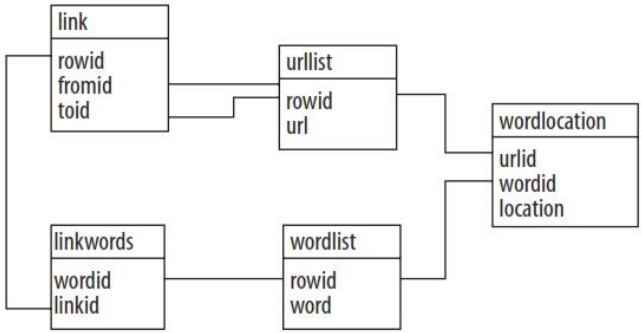

Ilustración 4-1 Esquema para el motor de búsqueda

Todas las tablas en SQLite tiene un campo llamado `rowid` por defecto, de forma que no se necesita especificar explícitamente un ID para estas tablas. Para crear una función para añadir todas las tablas, agregue este código al final del *searchengine.py* para ser parte de la clase `crawler`:

```
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( )
```

Esta función creará el esquema para todas las tablas que usarás, junto con algunos índices para agilizar la búsqueda. Estos índices son importantes, dado que el dataset puede ser muy largo. Ingresa estos comandos en tu sesión de Python para crear una base de datos llamada *searchindex.db*:

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

Encontrando las palabras en una página

Los archivos que has descargado desde la Web son HTML y estos contienen una gran cantidad de etiquetas, propiedades, y otra información que no pertenecen al índice. El primer paso es extraer todas las partes de la página que son texto. Puedes hacer esto buscando los nodos de texto y recolectando todo su contenido. Agrega este código a tu función `gettextonly`:

```
def gettextonly(self,soup):
      v=soup.string
      if v==None:
        c=soup.contents
        resulttext=''
      for t in c:
        subtext=self.gettextonly(t)
        resulttext+=subtext+'\n'
        return resulttext
      else:
        return v.strip( )
```

La función devuelve una cadena larga conteniendo todo el texto de la página. Hace esto recorriendo recursivamente el documento HTML, viendo los nodos de texto. El texto que estuvo en diferentes secciones se separa en párrafos diferentes. Es importantes preservar el orden de las secciones para algunas de las métricas que calcularás después.

La siguiente es la función `separatewords`, la cual divide una cadena en una lista de palabras separadas de forma que estas puedan ser agregadas al índice. No es tan fácil como podrías pensar el hacer esto perfectamente, y ha habido muchas investigaciones sobre cómo mejorar esta técnica. Sin embargo, para estos ejemplos es suficiente considerar todo lo que no es una letra o número como un separador. Puedes hacer esto usando expresiones regulares. Reemplaza la definición de `separatewords` con lo siguiente:

```
# Separa las palabras por un caracter que no sea espacio en blanco
def separatewords(self,text):
      splitter=re.compile('\\W*')
  return [s.lower( ) for s in splitter.split(text) if s!='']
```

Debido a que esta función considera todo lo que es no alfabético como un separador, no tiene problemas extrayendo palabras en inglés, pero no puede manejar apropiadamente términos como “C++” (aunque no tiene problemas para buscar Python). Puedes experimentar con la expresión regular para hacer que trabaje mejor para diferentes tipos de búsqueda.

Otra posibilidad es remover los sufijos de las palabras usando un algoritmo de derivación. Estos algoritmos intentan convertir las palabras a sus lexemas. Por ejemplo la palabra “indexing” se convierte en “index” de forma que a los que buscan la palabra “index” se le muestran también los documentos que contienen “indexing”. Para hacer esto, se derivan las palabras mientras se rastrea los documentos y también en la consulta de búsqueda. Una completa discusión del stemming está fuera del alcance de este libro, pero puedes encontrar información en *http://www.tartarus.org/~martin/PorterStemmer/index.html*.

Añadiendo al índice

Estas listo para completar en el código el método para addtoindex. Este método llamará las dos funciones que fueron defindas en la sección previa para obtener una lista de palabras en la página. Luego se añade la página y todas las palabras al índice, y se crearán enlaces entre ellos con sus localizaciones en el documento. Para este ejemplo, la localización será el índice dentro de la lista de palabras.

Aquí el código para `addtoindex`:

```
  # Indexa una página individual
  def addtoindex(self,url,soup):
    if self.isindexed(url): return
    print 'Indexing '+url
    # Get the individual words
    text=self.gettextonly(soup)
    words=self.separatewords(text)
    # Get the URL id
    urlid=self.getentryid('urllist','url',url)
    # Link each word to this url
    for i in range(len(words)):
      word=words[i]
      if word in ignorewords: continue
      wordid=self.getentryid('wordlist','word',word)
      self.con.execute("insert into wordlocation(urlid,wordid,location) \
        values (%d,%d,%d)" % (urlid,wordid,i))
```

Necesitas también esto para actualizar la función `getentryid`. Todo lo que esto hace es retornar el ID de una entrada. Si la entrada no existe, esta se crea y le ID es retornado:

```
  def getentryid(self,table,field,value,createnew=True):
    cur=self.con.execute(
    "select rowid from %s where %s='%s'" % (table,field,value))
```

```
    res=cur.fetchone( )
    if res==None:
      cur=self.con.execute(
      "insert into %s (%s) values ('%s')" % (table,field,value))
      return cur.lastrowid
    else:
      return res[0]
```

Finalmente, necesitas completar el código de `isindexed`, el cual determina si la página ya está en la base de datos, y también si hay alguna palabra asociada con esta:

```
  # Devuelve True si este URL ya esta indexado
  def isindexed(self,url):
    u=self.con.execute \
      ("select rowid from urllist where url='%s'" % url).fetchone( )
    if u!=None:
      # Check if it has actually been crawled
      v=self.con.execute(
      'select * from wordlocation where urlid=%d' % u[0]).fetchone( )
      if v!=None: return True
    return False
```

Ahora puedes correr el *crawler* y tenerlo actualizado con las páginas. Puedes hacer esto en una sesión interactiva:

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> pages= ['https://es.wikipedia.org/wiki/Python']
>> crawler.crawl(pages)
```

Si quieres asegurarte que el crawler trabaja apropiadamente, puedes intentar revisar las entradas para una palabra consultando la base de datos:

```
>> [row for row in crawler.con.execute('select rowid from wordlocation
where wordid=1')]
[(1,), (46,), (330,), (232,), (406,), (271,), (192,),...
```

La lista que se retorna es la lista de todos los IDs de los URLs conteniendo “Word”, lo cual significa que has corrido una búsqueda de texto completo satisfactoriamente. Este es un buen comienzo, pero esto solamente puede trabajar con una palabra a la vez, y devolverá los documentos en el orden en el que fueron cargados. La siguiente sección te mostrará cómo puedes expandir esta funcionalidad para hacer estas búsquedas con múltiples palabras en una consulta.

Consultando

Ahora tienes un crawler trabajando y una gran colección de documentos indexados, estás listo para configurar la parte de la búsqueda de un motor de búsqueda. Primero crear una nueva clase en *searchengine.py* que usarás para la búsqueda:

```
class searcher:
  def __init__(self, dbname):
    self.con = sqlite.connect(dbname)

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

La tabla `wordlocation` nos proporciona una manear facil de enlazar palabras a tablas, de manera que es muy fácil ver qué páginas contienen una palabra simple. Sin embargo, un motor de búsqueda está un poco limitado si no permite búsquedas de múltiples palabras. Para hacer esto, necesitas una función de consulta que tome una cadena de consulta, la divida en palabras separadas, y construya una consulta SQL para encontrar sol aquellas URLs que contienen todas las palabras. Agregue esta función a la definición para la clase `searcher`:

```
  def getmatchrows(self,q):
    # Cadenas para construir la consulta
    fieldlist='w0.urlid'
    tablelist=''
    clauselist=''
    wordids=[]
    # Dividir las palabras por los espacios
    words=q.split(' ')
    tablenumber=0
    for word in words:
        # Obtener el ID de la palabra
        wordrow=self.con.execute(
          "select rowid from wordlist where word='%s'" \
          % word).fetchone( )
        if wordrow!=None:
          wordid=wordrow[0]
          wordids.append(wordid)
          if tablenumber>0:
            tablelist+=','
            clauselist+=' and '
            clauselist+='w%d.urlid=w%d.urlid and ' \
                        % (tablenumber-1,tablenumber)
          fieldlist+=',w%d.location' % tablenumber
          tablelist+='wordlocation w%d' % tablenumber
          clauselist+='w%d.wordid=%d' % (tablenumber,wordid)
          tablenumber+=1
        # Crear la consulta a partir de las partes separadas
    fullquery='select %s from %s where %s'\
              % (fieldlist,tablelist,clauselist)
    cur=self.con.execute(fullquery)
    rows=[row for row in cur]
    return rows,wordids
```

Esta función luce un poco complicada, pero es solo la creación de una referencia a la tabla `wordlocation` para cada palabra en la lista y uniendo todas ellas por sus IDS de URL como se muestra en la Ilustración 4-2.

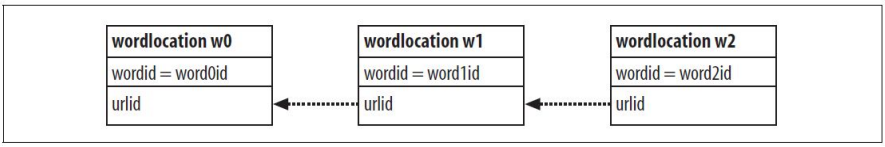

Ilustración 4-2 Tabla unida por getmachrows

Así, una consulta para dos palabras con los IDs 10 y 17 sería:

```
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
```

Pruebe llamando a esta función con tu primera consulta multi palabra:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.getmatchrows('functional programming')
([(1, 327, 23), (1, 327, 162), (1, 327, 243), (1, 327, 261),
(1, 327, 269), (1, 327, 436), (1, 327, 953),..
```

Habrás notado que cada ID de URL se devuelve muchas veces con diferentes combinaciones de localizaciones de palabras. En las siguientes secciones nos ocuparemos de algunas formas para rankear los resultados. El *ranking basado en contenidos* utiliza varias métricas posibles con el contenido de la página para determinar la relevancia de la consulta. El Ranking de enlaces entrantes usa la estructura de links del site para determinar su importancia. También exploraremos una forma de ver en qué hace click la gente cuando hacen búsquedas con el fin de mejorar el ranking a lo largo del tiempo.

Ranking basado en el contenido

Hasta ahora has conseguido recuperar páginas que cumplen con las consultas, pero el orden en el cual ellas son devueltas es simplemente el orden en el cual fueron rastreadas. En un conjunto grande de páginas, podrías quedar atascado entre una gran cantidad de contenidos irrelevantes para cualquier mención de cada una de los términos de la consulta para encontrar las páginas que realmente están relacionadas con tu búsqueda. Para solucionar este asunto, necesitas formas de dar un puntaje a las páginas un score para una consulta dada, asi como la habilidad de devolverlas que con el resultado del puntaje más alto primero.

Esta sección mostrará varias maneras de calcular un score basado sólo en la consulta y el contenido de la página. Estas métricas de scoring incluyen:

Frecuencia de Palabras

El número de veces que la palabra en la consulta aparece en el documento puede ayudar a determinar cuan relevante es el documento.

Localización del documento

El asunto principal de un documento probablemente aparece cerca del inicio del documento.

Distancia de palabra

Si hay múltiples palabras en la consulta, deben aparecer juntas o cerca en el documento.

Los motores de búsqueda primitivos a menudo trabajan con solo esos tipos de métricas y es posible entregar resultados útiles. En las secciones posteriores abarcamos formas de mejorar resultados con información externa a la página, tales como el número y la calidad de los link entrantes.

Primeramente necesitas un método nuevo que puede haga una consulta, obtenga las filas, las ponga en el diccionario y las muestre en una lista formateada. Agrega estas funciones a tu clase `searcher`:

```
  def getscoredlist(self,rows,wordids):
    totalscores=dict([(row[0],0) for row in rows])
    # aqui es donde pondrás después las funciones de score
    weights=[]
    for (weight,scores) in weights:
      for url in totalscores:
        totalscores[url]+=weight*scores[url]
    return totalscores

  def geturlname(self,id):
    return self.con.execute(
    "select url from urllist where rowid=%d" % id).fetchone( )[0]

  def query(self,q):
    rows,wordids=self.getmatchrows(q)
    scores=self.getscoredlist(rows,wordids)
    rankedscores=sorted([(score,url) for (url,score)
                         in scores.items( )],reverse=1)
    for (score,urlid) in rankedscores[0:10]:
      print '%f\t%s' % (score,self.geturlname(urlid))
```

Ahora el método de consulta no aplica ninguna puntuación para los resultados, pero este muestra las URLs junto con un marcador de posición para sus puntajes.

```
>> reload(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
...
```

La función más importante aquí es `getscoredlist`, la cual completarás a lo largo de esta sección. Así como agregas funciones de puntuación, puedes agregar llamadas a la lista `weights` (la línea en negrita) y empezar a obtener algunos puntajes reales.

Función de normalización

Todos los métodos introducidos aquí devuelven diccionarios de los IDs de las URLs y un puntaje numérico. Para complicar las cosas, algunas veces un puntaje alto es mejor y algunas veces un puntaje bajo también lo es. Con el fin de comparar los resultados de los diferentes métodos, necesitamos normalizarlos; esto es, ponerlos dentro del mismo rango y dirección.

La función de normalización tomará un diccionario de IDs y puntajes y devuelve un nuevo diccionario con los mismos IDs, pero con puntajes entre 0 y 1. Cada puntaje es escalado de acuerdo a qué tan cercano está del mejor resultado, el cual siempre tendrá un puntaje de 1. Todo lo que tienes que hacer es pasarle a la función una lista de puntajes e indicarle si es mejor un puntaje alto o uno bajo:

```
  def normalizescores(self,scores,smallIsBetter=0):
    vsmall=0.00001 # Evita la división por cero errores
    if smallIsBetter:
      minscore=min(scores.values( ))
      return dict([(u,float(minscore)/max(vsmall,l)) for (u,l) \
        in scores.items( )])
    else:
      maxscore=max(scores.values( ))
      if maxscore==0: maxscore=vsmall
      return dict([(u,float(c)/maxscore) for (u,c) in scores.items( )])
```

Cada vez que la función de putuación llama a esta función para normalizar esta devuelve un valor entre 0 y 1.

Frecuencia de palabras

La métrica de frecuencia de palabras puntúa una página basada en cuantas veces aparecen las palabras de la consulta en esta página. Si busco “Python”, obtendría una página sobre Python (o pitones) con muchas menciones de la palabra y no una página acerca de un músico quien ha mencionado que tiene una pitón como serpiente.

La función de frecuencia de palabras se ve así. Puedes adicionarla a la clase `searcher`:

```
  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)
```

Esta función crea un diccionario con una entrada para cada ID de URL único en las filas, y cuenta cuántas veces aparece cada ítem. Entonces normaliza los puntajes (el más grande es el mejor en este caso) y devuelve el resultado.

Para activar la puntuación de las frecuencias en tus resultados, cambie la línea de `weights` en `getscoredlist` para leer:

```
weights=[(1.0,self.frequencyscore(rows))]
```

Ahora puedes intentar otra búsqueda y ver qué tan bien trabaja como una métrica de puntuación:

```
>> reload(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
0.036394 http://kiwitobes.com/wiki/Programming_paradigm.html
...
```

Esto devuelve la página con “Functional Programming” en primer lugar, seguido por otras páginas relevantes. Note que “Functional programming” puntúa cuatro veces mejor que el resultado directamente debajo de este. La mayoría de motores de búsqueda no reportan puntajes a los usuarios finales, pero esos puntajes pueden ser muy útiles para algunas aplicaciones.

Localización de documentos

Otra métrica simpe para determinar la relevancia de una página es encontrar la localización de algunos términos en la página. Usualmente. Si una página es relevante para el término de búsqueda, este podría aparecer más cerca al principio de la página e incluso en el título. Para tomar ventaja de esto, el motor de búsqueda puede poner un puntaje alto si el término de consulta aparece antes en el documento. Afortunadamente para nosotros, cuando las páginas fueron indexadas antes, las ubicaciones de las palabras con grabadas, y el título de la página está primero en la lista.

Agrega este método al buscador:

```
  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)
```

Recuerde que el primer item en cada fila es el ID de la URL, seguida por la ubicación de todos los diferentes términos de búsqueda. Cada ID puede aparecer varias veces, una por cada combinación de ubicaciones. Para cada fila, el método suma las ubicaciones de todas las palabras y determina cómo este resultado se compara con el mejor resultado para esta URL hasta el momento. Entonces pasa los resultados finales a la función de `normalize`. Note que `smallIsBetter` significa que la URL con la suma de ubicaciones más baja obtiene un puntaje de 1.0.

Para ver que el resultado se muestre como si se usara solamente el puntaje de ubicación, cambia la línea `weights` por esto:

```
    weights=[(1.0,self.locationscore(rows))]
```

Ahora itentemos la consulta una vez más en su intérprete:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
```

Notarás que “Functional Programming” es aún el ganador, pero los otros mejores resultados son ahora ejemplos de lenguajes de programación funcional. La busca previa devuelve resultados en los cuales las palabras fueron mencionadas varias veces, pero esta tiende a ser discusiones sobre lenguajes de programación en general. Con esta búsqueda, sin embargo, la presencia de las palabras en la sentencia abierta (ej. “Haskell is a standarizaed pure functional programming language”) nos dará un puntaje mucho más alto.

Es importante resaltar que ninguna de las métricas mostradas hasta ahora es mejor todos los casos. Ambas listas son válidas dependiendo de los intentos del buscador, y diferentes combinaciones de pesos son requeridas para dar los mejores resultados para un conjunto particular de documentos y aplicaciones. Puedes intentar experimentar con diferentes pesos para las dos métricas cambiando la línea `weights` a algo como esto:

```
    weights=[(1.0,self.frequencyscore(rows)),
             (1.5,self.locationscore(rows))]
```

Experimenta con diferentes pesos y consultas, observa cómo tus resultados son afectados.

La ubicación es una métrica más difícil de engañar que la frecuencia de las palabras, dado que los autores de páginas pueden solo poner una palabra al principio de un documento y repetir esto no hace ninguna diferencia con los resultados.

Distancia de palabras

Cuando una consulta contiene múltiples palabras, es útil a menudo buscar resultados en los cuales las palabras en la consulta son cercanas a cada una de las otras en la página. La mayoría de veces, cuando la gente hace consultas de múltiples palabras, ellos están interesados en una página que relaciona conceptualmente las palabras diferentes. Esto es un poco más lento que la búsqueda de frases entre comillas soportada por la mayoría de motores de búsqueda donde las palabras deben aparecer en el orden correcto sin palabra adicionales – en este caso, la métrica puede tolerar un orden distinto y palabras adicionales entre las palabras consultadas.

Las función `distancescore` se ve muy similar a `locationscore`:

```
def distancescore(self,rows):
  # If there's only one word, everyone wins!
  if len(rows[0])<=2: return dict([(row[0],1.0) for row in rows])
  # Initialize the dictionary with large values
  mindistance=dict([(row[0],1000000) for row in rows])
  for row in rows:
    dist=sum([abs(row[i]-row[i-1]) for i in range(2,len(row))])
    if dist<mindistance[row[0]]: mindistance[row[0]]=dist
  return self.normalizescores(mindistance,smallIsBetter=1)
```

La principal diferencia es que cuando la función itera a través de las ubicaciones, esta toma las diferencias entre cada ubicación y la previa. Dado que cada combinación de distancias es devuelta por la consulta, está garantizado encontrar la distancia más pequeña.

Puedes probar la métrica de distancia de palabras por ti mismo si quieres, pero en realidad trabaja mejor si se combina con otras métricas. Prueba agregando distan escore a la lista de pesos y cambiar los números para ver como esto afecta al resultado de las consultas.

Ejercicios

1. Separación de palabras. El método `separatewords` actualmente considera cualquier carácter no alfabético como separador, lo que significa que no indexará apropiadamente entradas de índice como “C++”,”$20”,”Ph.D”, o “617-555-1212”. ¿Cuál es la mejor forma de separar palabras? ¿Funciona utilizar espacios en blanco cómo separador? Escriba una mejor función separadora de palabras.
2. Operaciones boolenas. Muchos motores de búsqueda soportan consultas booleanas, las cuales permiten a los usuarios construir búsquedas como “python OR perl”. Una búsqueda OR puede funcionar haciendo las consultas por separado y combinando los resultados, pero ¿Qué ocurre con “Python AND (program OR code)”? Modifique los métodos de consulta para soportar algunas operaciones booleanas básicas.
3. Coincidencias exactas. Los motores de búsqueda suelen soportar consultas de “coincidencia exacta”, donde las palabras en la página deben coincidir con la consulta en el mismo orden con ninguna palabra adicional entre ellas. Cree u a nueva versión de `getrows` que solamente devuelva resultados que son coincidencias exactas. (consejo. Puedes usar sustracción en SQL para obtener la diferencia entre las ubicaciones de las palabras).