

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

```
def addlinkref(sel f, url From, url To, linkText):  
    pass  
# Iniciando con una lista de páginas, hace una búsqueda  
# primero en anchura hasta la profundidad dada  
# indexando las páginas que lleguen  
def crawl(sel f, pages, depth=2):  
    pass  
# Crea las tablas de la base de datos  
def createindextables(sel f):  
    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 conjunto 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

Urllib2 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 Beautiful Soup, 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 `Beautiful Soup` 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 *searchengine.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 porción de localización
                    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, `indexed`, 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 de 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.conn.close()

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

Configuración del esquema

No corras el código justo aquí – aún necesitas preparar la base de datos. El esquema para la indexación básica es de cinco tablas. La primera tabla (`urlist`) 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 qué enlace. El esquema se muestra en la

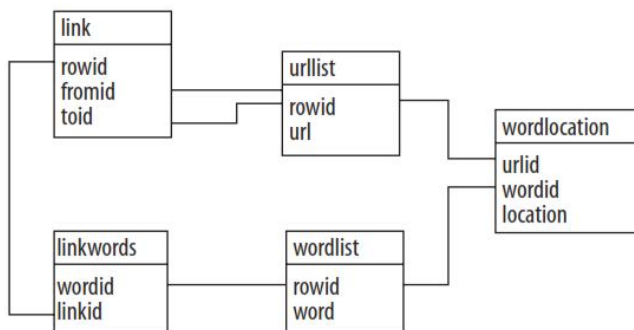


Ilustración 4-1 Esquema para el motor de búsqueda

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.conn.execute('create table urlist(url)')
    self.conn.execute('create table wordlist(word)')
    self.conn.execute('create table wordlocation(urlid, wordid, location)')
    self.conn.execute('create table link(fromid integer, toid integer)')
    self.conn.execute('create table linkwords(wordid, linkid)')
    self.conn.execute('create index wordidx on wordlist(word)')
    self.conn.execute('create index urlidx on urlist(url)')
    self.conn.execute('create index wordurlidx on wordlocation(wordid)')
    self.conn.execute('create index urltoidx on link(toid)')
    self.conn.execute('create index urlfromidx on link(fromid)')
```

```
sel f. 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.createindexes()
```

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(sel f, soup):
    v=soup.string
    if v==None:
        c=soup.contents
        resuttext=''
        for t in c:
            subtext=sel f.gettextonly(t)
            resuttext+=subtext+'\n'
        return resuttext
    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(sel f, 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 definidas 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(sel f, url, soup):
    if sel f.issindex(url): return
    print 'Indexing ' +url
    # Get the individual words
    text=sel f.gettextonly(soup)
    words=sel f.separatewords(text)
    # Get the URL id
    urlid=sel f.getentryid('urlist', 'url', url)
    # Link each word to this url
    for i in range(len(words)):
        word=words[i]
        if word in ignorewords: continue
        wordid=sel f.getentryid('wordlist', 'word', word)
        sel f.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(sel f, table, field, value, createnew=True):
    cur=sel f.con.execute(
        "select rowid from %s where %s='%s'" % (table, field, value))
```



```
res=cur.fetchone( )
if res==None:
    cur=sel f. 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(sel f, url):
    u=sel f. con. execute \
        ("select rowid from urllist where url='%s'" % url).fetchone( )
    if u!=None:
        # Check if it has actually been crawled
        v=sel f. 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)
```