INTELIGENCIA COLECTIVA

Tabla de contenido

Capítulo 1 Introducción a la inteligencia colectiva

Netflix es una compañía de renta de DVD que permite a la gente elegir películas que son enviadas a sus casas, y hace recomendaciones basadas en las películas que los clientes han rentado previamente. En el 2006 anunció un premio de un millón de dólares para la primera persona que mejorara la precisión de su sistema de recomendaciones en 10%. Cientos de equipos alrededor del mundo participaron y en abril de 2007, el equipo líder había logrado una mejora de 7%. Usando datos acerca de qué películas había disfrutado cada cliente, Netflix fue capaza de recomendar películas a otros consumidores que nunca habían oído ni mantenido contacto con ellos.

El motor de búsqueda Google empezó en 1998, en el tiempo cuando existían varios motores de búsqueda, y muchos asumían que un nuevo participante nunca podría ubicarse entre los gigantes. Los fundadores de Google, sin embargo, tomaron un enfoque completamente nuevo para rankear los resultados de las búsquedas de millones de sitios para decidir qué páginas eran las mas relevantes. Los resultados de Google fueron mucho mejores que los de los otros y en 2004 manejaban el 85% de las búsquedas en la Web. Sus fundadores están ahora entre las 10 personas mas ricas del mundo.

¿Que tienen estas compañías en común? Ellos crearon nuevas oportunidades de negocio usando algoritmos sofisticados para combinar datos recolectados de muchas personas distintas. La habilidad para recolectar información y el poder computacional para interpretarlo ha permitido grandes oportunidades de colaboración y un mejor entendimiento de los usuarios y los consumidores. Esta forma de trabajo está en todo lugar – los sites de datos quieren ayudar a la gene a encontrar sus mejores elecciones mas rápidamente, compañías que predicen cambios en los precios de los pasajes de avión, etc.

Son algunos ejemplos en el excitante campo de la inteligencia colectiva, y la proliferación de nuevos servicios significa que hay nuevas oportunidades apareciendo cada día. Creo que el entendimiento del aprendizaje automático y los métodos estadísticos llegarán a ser mas importantes en una amplia variedad de campos, pero particularmente en interpretar y organizar la vasto volumen de información que está siendo creado por la gente de todo el mundo.

Lo que es la inteligencia colectiva

La gente ha usado el termino Inteligencia Colectiva por décadas, y esta ha llegado a ser popular y mas importante con el advenimiento de las nuevas tecnologías de comunicación. La expresión puede llevar a pensar en conciencia grupal o en fenómenos paranormales, cuando los tecnólogos utilizan esta frase usualmente

significa la combinación del comportamiento, preferencias, o ideas de un grupo de gente para crear nuevo entendimiento.

La inteligencia colectiva era posible antes del Internet. No necesitas una página Web para recolectar datos de grupos dispares de gente, combinarlos y analizarlos. Una de las formas más básicas de hacer eso es hacer una encuesta o un censo. Recolectar respuestas de un gran grupo de gente permite sacar conclusiones estadísiticas acerca del grupo que los elementos individuales del grupo no hubieran conocido de ellos mismos. Construir nuevas conclusiones a partir de las contribuciones individuales es en realidad lo que la inteligencia colectiva es.

Un ejemplo bien conocido es el mercado financiero, donde un precio no se da por un esfuerzo individual o un esfuerzo combinado, pero si por el comportamiento de negocio de mucha gene independiente actuando en lo que ellos creen que es su propio interés. Esto parece contraintuitivo al principio, pero los mercados en los cuales los participantes hacen contratos basados en sus creencias…

Capítulo 2
Haciendo Recomendaciones

Para empezar vamos a mostrar las formas en las que se pueden utilizar las preferencias de un grupo de gente para hacer recomendaciones a otras personas. Hay muchas aplicaciones para este tipo de información, tales como hacer recomendaciones de productos para compras online, sugerir sitios web interesantes o ayudar a la gente a encontrar música y películas.

Las preferencias pueden ser recolectadas en muchas formas. Algunas veces los datos son ítems que la gente ha pedido, y las opiniones sobre esos ítems se pueden representar como votos o escalas. En este capítulo analizaremos diferentes formas de representar estos casos de forma que todas ellas trabajan con el mismo conjunto de algoritmos, y podamos crear ejemplos con críticas de cine y *bookmarking* social.

Filtros colaborativos

Una forma poco técnica de obtener recomendaciones sobre productos, películas o entretenimiento es preguntarles a los amigos. También se sabe que algunos de los amigos tienen mejor gusto que otros y eso lo aprendemos viendo si ellos tienen gustos similares a los nuestros.

Puede ser poco práctico decidir preguntando a un pequeño grupo de personas, dado que ellos pueden no haber probado todas las opciones. Para esto se desarrollaron un conjunto de técnicas llamadas filtrado colaborativo.

Un algoritmo de filtrado colaborativo trabaja buscando en un gran grupo de gente y encontrando pequeños conjuntos con los mismos gustos que los tuyos. Esto analiza otras cosas que a ellos les gustan y los combina para crear una lista de sugerencias rankeadas. Hay varias formas de decidir cuales personas son similares combinando sus opciones para hacer una lista; este capítulos abarca algunas de ellas.

Recopilando preferencias

Lo primero que necesitas es una forma para representar diferentes personas y sus preferencias. En Python, una forma simple de hacerlo es usar un diccionario anidado. Si quieres trabajar a través del ejemplo en esta sección, crea un archivo llamado *recommendatios.py*, e inserta el siguiente código para crear el conjunto de datos.

```
# un diccionario de criticas de cine y sus puntajes
# de un pequeno conjunto de peliculas
critics={
    'Lisa Rose': {
```

```
    'Lady in the Water': 2.5,
    'Snakes on a Plane': 3.5,
    'Just My Luck': 3.0,
    'Superman Returns': 3.5,
    'You, Me and Dupree': 2.5,
    'The Night Listener': 3.0
    },
  'Gene Seymour': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 3.5,
    'Just My Luck': 1.5,
    'Superman Returns': 5.0,
    'The Night Listener': 3.0,
    'You, Me and Dupree': 3.5
    },
  'Michael Phillips': {
    'Lady in the Water': 2.5,
    'Snakes on a Plane': 3.0,
    'Superman Returns': 3.5,
    'The Night Listener': 4.0
    },
  'Claudia Puig': {
    'Snakes on a Plane': 3.5,
    'Just My Luck': 3.0,
    'The Night Listener': 4.5,
    'Superman Returns': 4.0,
    'You, Me and Dupree': 2.5
    },
  'Mick LaSalle': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 4.0,
    'Just My Luck': 2.0,
    'Superman Returns': 3.0,
    'The Night Listener': 3.0,
    'You, Me and Dupree': 2.0
    },
  'Jack Matthews': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 4.0,
    'The Night Listener': 3.0,
    'Superman Returns': 5.0,
    'You, Me and Dupree': 3.5
    },
  'Toby': {
    'Snakes on a Plane':4.5,
    'You, Me and Dupree':1.0,
    'Superman Returns':4.0
    }
  }
```

Trabajaremos interactivamente en este capítulo de manera que debes guardar recommendations.py en un lugar donde el intérprete pueda encontrarlo. Esto puede

ser en el directorio *Python/Lib*, pero la forma más fácil de hacer esto es iniciar el intérprete de Python en el mismo directorio en el cual has guardado el archivo.

Este diccionario usa un ranking de 1 a 5 para expresar gustan cada uno de estos críticos de una película dada. Necesitas expresar estas preferencias en una forma numérica. Si estas construyendo un sitio de ventas, podrías usar un valor de 1 para indicar si alguno ha comprado un ítem en el pasado y un valor de 0 para indicar que no lo han comprado. Para un *site* donde las personas votan sobre nuevos productos se pueden utilizar valores de -1, 0 y 1 para representar que "no le gusta", "no opina" y "le gusta" como se muestra en la tabla.

Tickets de concierto		Compras Online		Recomendaciones de sitios	
Compró	1	Compró	2	Le gusta	1
No compró	0	Lo buscó	1	No vota	0
		No compró	0	No le gusta	-1

El uso de un diccionario es conveniente para experimentar con los algoritmos y para propósitos ilustrativos. Es fácil buscar y modificar el diccionario. Inicia tu intérprete de Python y prueba los siguientes comandos:

```
c:\code\collective\chapter2> python
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>
>> from recommendations import critics
>> critics['Lisa Rose']['Lady in the Water']
2.5
>> critics['Toby']['Snakes on a Plane']=4.5
>> critics['Toby']
{'Snakes on a Plane':4.5,'You, Me and Dupree':1.0}
```

Encontrando usuarios similares

Después de recolectar datos sobre las cosas que les gustan a las personas, necesitas una manera de determinar cuan similares son en cuanto a sus gustos. Haces esto comparando cada persona con cada una de las otras y calculando una *puntación de similitud*. Hay unas cuantas maneras de hacer esto y en esta sección se mostrarán dos sistemas para calcular puntajes de similitud: la *distancia Eucl*idiana y la *correlación de Pearson*

Calificación de la distancia euclidiana

Una forma muy simple de calcular un puntaje de similitud es usar la distancia euclidiana, el cual toma los ítems que las personas han calificado en común y usa

estos como ejes de una gráfica. Se puede entonces ubicar a las personas en el gráfico y ver qué tan cerca está. Como se muestra en la figura.

Ejm.

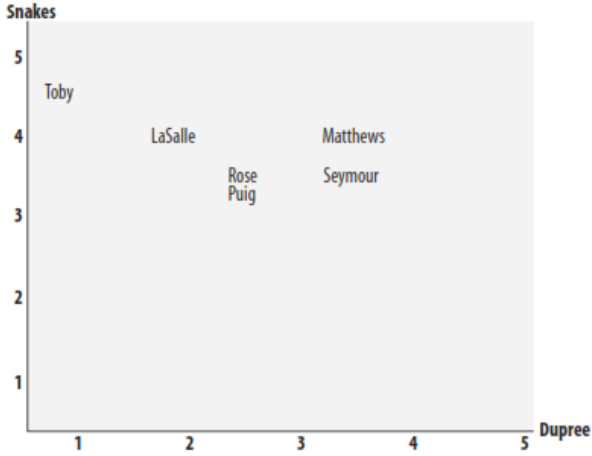

Ilustración 2-1 Personas en el espacio de preferencias

Esta figura muestra a las personas en el espacio de preferencias. *Toby* ha sido graficado a 4.5 en el eje de *Snakes* y a 1.0 en el eje de *Dupree*. Mientras más cercanas estén dos personas en el espacio de preferencias, sus gustos son más similares. Porque el gráfico es de dos dimensiones, puedes solo ver dos rankings al mismo tiempo, pero el principio el mismo para grandes conjuntos de rankings.

Para calcular la distancia entre *Toby* y *LaSalle* en el gráfico, se toma la diferencia entre cada eje al cuadrado y se suman estas, entonces se saca la raíz cuadrada de la suma. En Python, se puede usar la función `pow(n,2)` para elevar al cuadrado un número y para sacar la raíz cuadrada con la función `sqrt(n)`.

```
>> from math import sqrt
>> sqrt(pow(5-4,2)+pow(4-1,2))
3.1622776601683795
```

Esta fórmula calcula la distancia, la cual será pequeña para las personas que son más similares. Sin embargo, necesitamos una función que nos de valores altos para personas quienes son similares. Esto puede ser hecho dividiendo 1 entre el valor obtenido:

```
>> 1/(1+sqrt(pow(5-4,2)+pow(4-1,2)))
0.2402530733520421
```

Esta nueva función siempre retorna un valor entre 0 y 1, donde un valor de 1 significa que dos personas tiene idénticas preferencias. Podemos poner todo esto junto para crear una función para calcular la similitud. Adicionando el siguiente código a *recommentations.py:*

```
from math import sqrt

# Retorna un puntaje de similardiad basado en distancia
# para persona1 y persona2
def sim_distance(prefs,person1,person2):
  # Obtiene la lista de los items similares
  si={}
  for item in prefs[person1]:
    if item in prefs[person2]:
      si[item]=1
  # si no tienen cosas en comun retorna 0
  if len(si)==0: return 0
  # susma los cuadrados de las diferencias
  sum_of_squares=sum([pow(prefs[person1][item]-prefs[person2][item],2)
                      for item in prefs[person1]
                      if item in prefs[person2]])
  if sum_of_squares == 0: return 1
  return 1.0/(1+sqrt(sum_of_squares))
```

Esta función puede ser llamada con dos nombres para obtener un puntaje de similitud. En tu intérprete de Python corre lo siguiente:

```
>> reload(recommendations)
>> recommendations.sim_distance(recommendations.critics,
...   'Lisa Rose','Gene Seymour')
0.148148148148
```

Esto da una similitud entere *Lisa Rose* y *Gene Seymour*. Intente esto con otros nombres para ver si puedes encontrar gente que tiene más o menos e común.

Puntaje de correlación de Pearson

Una forma más sofisticada para determinar la similitud entre intereses de personas es usando el coeficiente de correlación de Pearson. El coeficiente de correlación es una medida de cuan bien dos conjuntos de datos se ajustan a una línea recta. La fórmula para esto es más complicada que la distancia euclidiana, pero tiende a dar mejores resultados en situaciones donde los datos no están bien normalizados – por ejemplo, si el ranking de un crítico es rutinariamente más duro que el resto.

Para visualizar este método, puedes dibujar los puntajes de los críticos en una gráfica, como se muestra en la Ilustración 2-2.

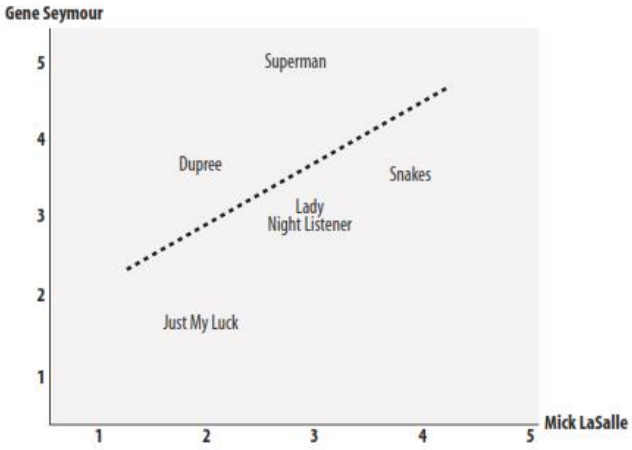

Ilustración 2-2 Comparando dos críticos de películas en un gráfico de dispersión

Puedes ver también una línea recta en la gráfica. Esta es llamada el mejor ajuste lineal porque es la más cercana cómo es posible a todos los ítems. Si dos críticos tienen puntajes idénticos para todas las películas esta línea se verá diagonal y tocará a cada ítem en la gráfica dando una correlación perfecta de 1. En el caso ilustrado, los críticos están en desacuerdo en unas pocas películas de manera que el puntaje de correlación es de aproximadamente 0.4 la Ilustración 2-3

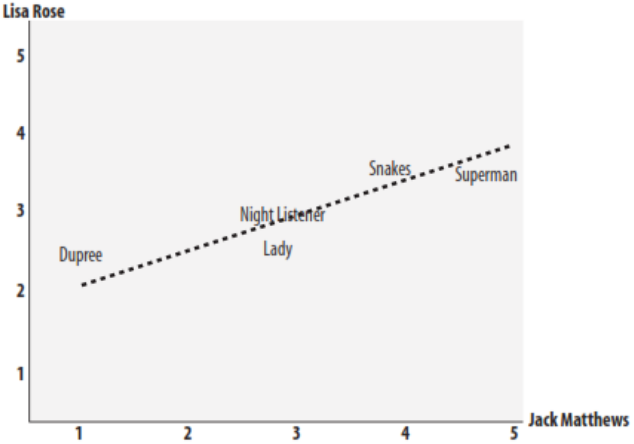

Ilustración 2-3 Dos críticos con un alto puntaje de correlación

Un aspecto interesante del uso del puntaje de Pearson, es que es que corrige el grado de inflación. En esta figura, *Jack Matthews* tiende a dar más alto puntaje que Lisa

Rose, pero la línea aún encaja porque ellos tienen relativamente preferencias similares. Si un crítico se inclina a dar puntajes altos que otro, esto puede ser aun una correlación perfecta si las diferencias entre sus puntajes son consistentes. La distancia euclidiana vista antes puede decir que dos críticos son distintos porque uno es considerablemente más duro que otro, aun si sus gustos son similares. Dependiendo de su aplicación particular, este comportamiento puede o no puede ser lo que quiere.

El código para la correlación de Pearson primero encuentra los ítems calificados por ambos críticos. Entonces se calculan las sumas y la suma de los cuadrados de las calificaciones de los dos críticos y calcula la suma de los productos de sus calificaciones. Finalmente, se usan estos resultados para calcular el coeficiente de correlación de Pearson que se muestra en negrita en el código. A diferencia de la métrica de distancia, esta fórmula no es muy intuitiva, pero nos dice cuánto las variables cambian juntas divididas por el producto de cuánto varían individualmente.

Para usar esta fórmula, crea una nueva función con la misma forma de sim_distance en *recommendations.py:*

```
# Devuelve el coeficiente de correlacion de Pearson
def sim_pearson(prefs,p1,p2):
   # Obtiene la lista de items en comun
   si={}
   for item in prefs[p1]:
      if item in prefs[p2]:
         si[item]=1
   # Encontrar el numero de elementos
   n=len(si)
   # Si no hay items en comun retorna 0
   if n==0: return 0
   # Suma todas las preferencias de cada persona
   sum1=sum([prefs[p1][it] for it in si])
   sum2=sum([prefs[p2][it] for it in si])
   # Suma de los cuadrados
   sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
   sum2Sq=sum([pow(prefs[p2][it],2) for it in si])
   # Suma de los productos
   pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])
   # Calcula el coeficiente de pearson
   num=pSum-(sum1*sum2/n)
   den=sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
   if den==0: return 0
   r=num/den
   return r
```

Una forma más intuitiva de calcular el coeficiente de Pearson es:

```
def sim_pearson2(prefs, p1, p2):
   # Obtiene la lista de items en comun
   si={}
```

```
    for item in prefs[p1]:
      if item in prefs[p2]: si[item]=1
    # Encontrar el numero de elementos
    n=float(len(si)) #float es necesario para que la division sea real
    # Si no hay items en comun retorna 0
    if n==0: return 0
    mediaP1 = sum([prefs[p1][it] for it in si])/n
    mediaP2 = sum([prefs[p2][it] for it in si])/n
    devP1 = sqrt(sum([pow(prefs[p1][it]-mediaP1,2) for it in si])/(n-1))
    devP2 = sqrt(sum([pow(prefs[p2][it]-mediaP2,2) for it in si])/(n-1))
    covP1P2 = sum([(prefs[p1][it]-mediaP1)*(prefs[p2][it]-mediaP2)
              for it in si])/(n-1)
    r = covP1P2/(devP1*devP2)
    return r
```

Esta es más similar a la definición matemática:

$$\rho_{X,Y} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y}$$

$$\rho_{X,Y} = \frac{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i-\bar{y})^2}}$$

$$\rho_{X,Y} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2(y_i-\bar{y})^2}}$$

Donde:

σ_{XY} es la covarianza de (X, Y)

σ_X es la desviación típica de la variable X

σ_Y es la desviación típica de la variable Y

Esta función retornará un valor entre -1 y 1. Un valor de 1 significa que las dos personas tienen exactamente las mismas calificaciones para cada ítem. A diferencia de la métrica de distancia, tú no necesitas cambiar este valor para darle la correcta escala. Ahora puedes intentar obtener la correlación para la Ilustración 2-3.

¿Qué métrica de similitud deberías usar?

Hemos introducido funciones para dos métricas diferentes, pero hay actualmente muchas más formas de medir la similitud entre dos conjuntos de datos. La mejor de ellas dependerá de la aplicación, y vale la pena probarlas para ver cuál de ellas da los mejore resultados.

Las funciones en el resto del capítulo tienen un parámetro opcional de similitud, el cual apunta a una función para hacer más fácil experimentar: especificando

sim_pearson o sim_vector para elegir que parámetro de similitud usar. Hay muchas otras funciones como el coeficiente de Jaccard o la Distancia de Manhattan que puedes usar como función de similitud siempre que ellos reciban los mismos parámetros y retornen un float donde un valor alto significa similar.

Rankeando las críticas

Ahora que tenemos funciones para comparar dos personas, podemos crear una función que puntúe a todos contra cada una persona dada y encuentre a los más cercanos. En este caso, estoy interesado en aprender que crítico tiene gustos similares a los míos de forma que sepa cual sugerencia tomar cuando estoy decidiendo por una película. Agregue esta función a recommendations.py para obtener una lista ordenada de personas con gustos similares para una persona específica.

```
# Devuelve el mejor par para una persona desde el diccionario
# Numero de resultados y funcion de similaridad son parametros opcionales
def topMatches(prefs,person,n=5,similarity=sim_pearson):
   scores=[(similarity(prefs,person,other),other)
         for other in prefs if other!=person]
   # Ordena la lista de forma que el puntaje mas alto aparece primero
   scores.sort( )
   scores.reverse( )
   return scores[0:n]
```

esta función utiliza una compresión de listas de Python para comparar a Toby con cada uno de los otros usuarios usando una de las previamente definidas métricas de distancia. Esta retorna los primeros n ítems en una lista ordenada.

Llamando a esta función con el nombre de 'Toby' se obtiene una lista de críticas de películas y sus puntajes de similitud.

```
>> reload(recommendations)
>> recommendations.topMatches(recommendations.critics,'Toby',n=3)
[(0.99124070716192991, 'Lisa Rose'), (0.92447345164190486, 'Mick LaSalle'),
(0.89340514744156474, 'Claudia Puig')]
```

De esto se sabe que *Lisa Rose* tiene un gusto similar al mío. Si has visto alguna de estas películas podrías intentar agregarte al diccionario con tus preferencias y ver quien debería ser tu crítico favorito.

Recomendando Ítems

Encontrar una buena crítica para leer es bueno, pero lo que queremos es un recomendación de una película. Podríamos ver a la persona que ha tenido gustos similares a los míos y ver una película que le guste y que no haya visto aun *Toby*,

pero esto podría ser muy permisivo. Podría decidir ver lo que alguien muy parecido vio, pero puede que esta película tenga malas calificaciones de los otros críticos.

Para resolver esto, necesitas puntuar los ítems produciendo un puntaje con pesos que rankee las críticas. Tomar los votos de todas las demás críticas y multiplicar por cuan similares son a Toby los puntajes que ellos han dado para cada película. La tabla muestra cómo trabaja este proceso. Las similitudes menores que cero se consideran como 'cero'.

Critic	Similarity	Night	S.xNight	Lady	S.xLady	Luck	S.xLuck
Rose	0.99	3.0	2.97	2.5	2.48	3.0	2.97
Seymour	0.38	3.0	1.14	3.0	1.14	1.5	0.57
Puig	0.89	4.5	4.02			3.0	2.68
LaSalle	0.92	3.0	2.77	3.0	2.77	2.0	1.85
Matthews	0.66	3.0	1.99	3.0	1.99		
Total			12.89		8.38		8.07
Sim. Sum			3.84		2.95		3.18
Total/Sim. Sum			3.35		2.83		2.53

Esta tabla muestra los puntajes de correlación y las calificaciones que ellos les han dado a las tres películas. Las columnas que empiexan con S.x están multiplicadas por el rating, de forma que una persona quien es más similar a mi contribuirá más al puntaje total que una persona que es diferente a mi. La fila Total muestra la suma de todos estos números.

Podrías usar los totales para calcular los rankings, pero entonces una película revisada por más gente podría tener una gran ventaja. Lo correcto para esto es que se necesita dividir entre la suma de todas las similitudes para los críticos que han revisado esa película.

El código para esto bastante simple y trabaja con ambas, la distancia euclidiana o la correlación de Pearson. Agregue esto a *recommendations.py*

```
# Obteniendo recomendaciones para una persona usando un promedio
# con pesos de cada un de los rankings de los usuarios
def getRecommendations(prefs,person,similarity=sim_pearson):
  totals={}
  simSums={}
  for other in prefs:
    # evita que me compare a mi mismo
    if other==person: continue
    sim=similarity(prefs,person,other)
    # ignora puntajes menores que cero
    if sim<=0: continue
    for item in prefs[other]:
      # solamente putua películas que no he visto
      if item not in prefs[person] or prefs[person][item]==0:
```

```
            # Similitud * Score
            totals.setdefault(item,0)
            totals[item]+=prefs[other][item]*sim
            # Sum de similitudes
            simSums.setdefault(item,0)
            simSums[item]+=sim
    # Crea la lista normalizada
    rankings=[(total/simSums[item],item)
            for item,total in totals.items( )]
    # Retorna la lista ordenada
    rankings.sort( )
    rankings.reverse( )
    return rankings
```

El código itera sobre cada persona en el diccionario de preferencias. En cada caso, se calcula cuan similares son a una persona específica.

```
>> reload(recommendations)
>> recommendations.getRecommendations(recommendations.critics,'Toby')
[(3.3477895267131013, 'The Night Listener'), (2.8325499182641614, 'Lady in the Water'),
(2.5309807037655645, 'Just My Luck')]
>> recommendations.getRecommendations(recommendations.critics,'Toby',
...   similarity=recommendations.sim_distance)
[(3.5002478401415877, 'The Night Listener'), (2.7561242939959363, 'Lady in the Water'),
(2.4619884860743739, 'Just My Luck')]
```

No solo se obtiene un ranking de películas, sino también un sugerencia de cuál sería mi calificación para cada película.

Has construido un sistema de recomendaciones, el cual trabaja con cualquier tipo de productos. Todo lo que tienes que hacer es definir un diccionario de gente, ítems y puntajes, puedes usar esto para crear recomendaciones para cualquier persona.

Emparejando productos

Ahora que sabes cómo encontrar personas similares y recomendar productos para un persona dada, pero ¿Y si quieres ver qué productos son similares a otros? Puedes encontrar esto en páginas web de tiendas, en especial cuando el site no ha recopilado información acerca de ti.

En este caso puedes determinar la similitud observando a quien le gusta un ítem en particular y viendo las otras cosas que a ellos les gusta. Este es actualmente el mismo método nosotros usamos anteriormente para determinar la similitud entre las personas – necesitas intercambiar las personas y los ítems. Puedes usar los mismos métodos que ya escribiste antes si se transforma el diccionario de:

```
{'Lisa Rose': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.5},
'Gene Seymour': {'Lady in the Water': 3.0, 'Snakes on a Plane': 3.5}}
```

En:

{'Lady in the Water':{'Lisa Rose':2.5,'Gene Seymour':3.0},
'Snakes on a Plane':{'Lisa Rose':3.5,'Gene Seymour':3.5}} etc..

Agrega una función a recomendations.py para hacer esta transformación:

```
# Invierte el diccionario de forma que los items tendrán ahora una
# lista de criticos
def transformPrefs(prefs):
   result={}
   for person in prefs:
      for item in prefs[person]:
         result.setdefault(item,{})
         # Intercambia items y personas
         result[item][person]=prefs[person][item]
   return result
```

Entonces llamas a la función topMatches utilizada antes para encontrar el conjunto de películas más parecidas a *Superman Returns*:

```
>> reload(recommendations)
>> movies=recommendations.transformPrefs(recommendations.critics)
>> recommendations.topMatches(movies,'Superman Returns')
[(0.657, 'You, Me and Dupree'), (0.487, 'Lady in the Water'), (0.111, 'Snakes on a Plane'), (-0.179, 'The
Night Listener'), (-0.422, 'Just My Luck')]
```

Note que en este ejemplo existen algunos puntajes de correlación negativos, lo cual indica que aquellos a quienes les gusta Superman Returns tiende a desagradarles Just My Luck, como se muestra en la Ilustración 2-4.

Para enredar las cosas aún más, puedes obtener recomendaciones de críticos para una película. ¿Quizás estés tratando de decidir a quién invitar a una premiere?

```
>> recommendations.getRecommendations(movies,'Just My Luck')
[(4.0, 'Michael Phillips'), (3.0, 'Jack Matthews')]
```

No queda claro que intercambiar personas e ítems dará resultados útiles, pero en muchos casos esto puede permitir hacer comparaciones interesantes. Un vendedor pude recopilar historiales de ventas para recomendar productos a sus vendedores. Invertir los productos con las personas como hicimos aquí les permitiría buscar personas que podrían comprar ciertos productos. Esto puede ser muy útil para planificar una campaña de márketing para resaltar algunos ítems. Otro uso potencial es asegurarse que los nuevos links o recomendaciones son vistos por las personas a quienes les agrade más.

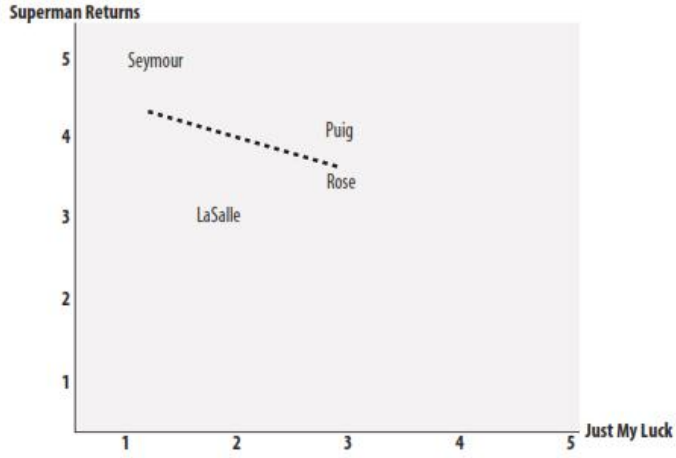

Ilustración 2-4 Superman Returns y Just My Luck tienen una correlación negativa

Filtrado basado en ítems

La forma en que el motor de recomendaciones ha sido implementado requiere el uso de todos los rankings de cada usuario para crear un dataset. Esto puede trabajar bien para unos pocos cientos de personas o ítems, pero un site muy grande como Amazon tiene millones de clientes y productos – comparando a un usuario con cada uno de los otros y comparando cada producto que cada usuario ha calificado puede ser muy lento. Además un site que vende millones de productos puede tener una muy pequeña superposición entre personas, lo cual puede hacer difícil decidir qué personas son similares.

La técnica que hemos utilizado se llama Filtrado Colaborativo Basado en el Usuario. Una alternativa es conocida como el Filtrado Colaborativo Basado en los Items. En los casos con conjuntos de datos muy grandes, el filgrado basado en ítems puede dar mejores resultados y permite que muchos de los cálculos sean realizados por adelantado de forma que si un usuario necesita recomendaciones estas pueden ser obtenidas más rápidamente.

El procedimiento para el filtrado basado en ítems se parece mucho al que ya hemos visto. La técnica general es precalcular los ítems más similares para cada ítem. Entonces, cuando deseas hacer recomendaciones a un usuario, observas en sus ítems mejor calificados y creas una lista de ítems más similares a ellos. La diferencia más importante es que, la comparación entre ítems no cambia tan a menudo como cambia la comparación entre usuarios. Esto significa que no tienes que calcular continuamente cada ítem más similar – puedes hacer esto unas cuantas veces o en una computadora aparte de su aplicación principal.

Construyendo el conjunto de datos de comparación de ítems

Para comparar ítems, lo primero que necesitas hacer es escribir una función que construya el conjunto de datos completo de ítems similares. Esto no tiene que ser hecho cada vez que las recomendaciones son necesarias, sino que construyes tu conjunto de datos una vez y reutilzas este cada vez que lo necesitas.

Para generar el conjunto de datos agrega la siguiente función a *recommendations.py*:

```
#construye el conjunto de datos para items similares
def calculateSimilarItems(prefs,n=10):
  # Crea un diccionario de itemas mostrando que otros items
  # son similares a ellos
  result={}
  # Invirte la matriz de preferencias para que se centre en los items
  itemPrefs=transformPrefs(prefs)
  c=0
  for item in itemPrefs:
    # Actualizaciones de estado para grandes conjuntos de datos
    c+=1
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # Encuentra los items mas similares a uno
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    result[item]=scores
  return result
```

Esta función primero invierte el diccionario de puntajes usando la función transformPrefs definida antes, dando una lista de ítems con datos de cómo ellos fueron calificados por cada usuario. Entonces itera sobre cada ítem y pasa el diccionario transformado a la función topMatches para obtener los ítems más similares con sus puntajes de similitud. Finalmente crea y devuelve un diccionario de ítems con una lista de sus ítems más similares.

En tu sesión de Python, construye el conjunto de datos de similitudes y ve que luce así:

```
>> reload(recommendations)
>> itemsim=recommendations.calculateSimilarItems(recommendations.critics)
>> itemsim
{'Lady in the Water': [(0.40000000000000002, 'You, Me and Dupree'),
                       (0.2857142857142857, 'The Night Listener'),...
 'Snakes on a Plane': [(0.22222222222222221, 'Lady in the Water'),
                       (0.18181818181818182, 'The Night Listener'),...
etc.
```

Recuerda que esta función solamente debe correrse lo suficientemente frecuente para mantener actualizadas las similitudes entre ítems. Necesitas hacer esto más a menudo al principio cuando la base de usuarios y número de calificaciones es pequeña, pero a medida que la base de usuarios aumenta, los puntajes de similitud entre ítems serán mas estables.

Obteniendo recomendaciones

Ahora estamos listos para dar recomendaciones usando el diccionario de similitud sin recorrer el conjunto de datos entero. Debes obtener todos los ítems que el usuario ha calificado, encontrar los ítems similares, y ajustar sus pesos de acuerdo a cuan similares son ellos.

La tabla muestra el proceso de encontrar recomendaciones usando el enfoque basado en ítems. A diferencia del anterior los críticos no están involucrados y en su lugar tenemos una matriz de películas que he calificado versus películas que no he calificado.

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Cada fila tiene una película que ya he visto, con mi calificación personal para esta. Para cada película que no he visto hay una columna que muestra que tan similares son estas a las películas que ya he visto – por ejemplo, el puntaje de similitud entre *Superman* y *The Night Listener* es 0.103. La columna rotulada con R.x muestra mi calificación de la película multiplicada por la similitud - como yo he calificado *Superman* con 4.0, el valor siguiente de *Night* en la fila de *Superman* es 4.0 (0.103) = 0.412.

El total de las filas muestra el total de los puntajes de similitud y el total de las columnas R.x para cada película. Para predecir cuál será mi calificación para cada película, dividiré el total de la columna R.x por el total de la columna de similitud. Mi calificación predicha para *The Nighth Listener* es 1.378/0.433 = 3.183.

Puedes usar esta funcionalidad adicionando una última función a *recommendations.py*:

```
def getRecommendedItems(prefs,itemMatch,user):
    userRatings=prefs[user]
    scores={}
    totalSim={}
    # Itera sobre los itemas calificados por el usuario
    for (item,rating) in userRatings.items( ):
        # Itera sobre los items similares a estos
        for (similarity,item2) in itemMatch[item]:
            # Ignorar si el usuario ha calificado ya este item
            if item2 in userRatings: continue
            # Suma ponderada de calificaciones de similitud
            scores.setdefault(item2,0)
```

```
        scores[item2]+=similarity*rating
        # Suma de todas las similitudes
        totalSim.setdefault(item2,0)
        totalSim[item2]+=similarity
    # Divide cada puntaje total por el total ponderado
    rankings=[(score/totalSim[item],item)
            for item,score in scores.items( )]
    # Devuelve el ranking del más alto al más bajo
    rankings.sort( )
    rankings.reverse( )
    return rankings
```

Puedes probar esta función con el conjunto de datos de similitudes que has construido en antes para obtener recomendaciones para Toby:

```
>> reload(recommendations)
>> recommendations.getRecommendedItems(recommendations.critics,
...itemsim,'Toby')
[(3.182, 'The Night Listener'),
 (2.598, 'Just My Luck'),
 (2.473, 'Lady in the Water')]
```

The Night Listener permanece en primer lugar con un margen significativo, y *Just My Luck* y *Lady in the Wather* han cambiado de lugar aunque ellos aún permanecen muy juntos. Lo más importante, la llamada a getRecommendedItems no tiene que calcular los puntajes de similitud para todos los otros críticos porque el conjunto de datos de similitud fue construido por adelantado.

Usando el Dataset de MovieLens

Para el ejemplo final, vamos a ver un dataset real de movie rating llamado MovieLens. MovieLens desarrollada por el proyecto GroupLens en la Universidad de Minnesota. Puedes descargar el dataset de *http://grouplens.org/datasets/movielens/*. Descargue el dataset en formato zip o tar.gz.

El archivo contiene varios archivos, pero algunos de los que nos interesa son u.item, el cual contiene una lista de IDs de películas y títulos, y u.data, el cual contiene las calificaciones actuales en este formato.

```
196  242  3    881250949
186  302  3    891717742
22   377  1    878887116
244  51   2    880606923
166  346  1    886397596
298  474  4    884182806
```

Cada línea tiene un ID de usuario, un ID de una película, la calificación dada por el usuario y un registro de tiempo. Puedes obtener los títulos de las películas, pero los

usuarios son anónimos, por lo que trabajarás con los IDs de usuario en esta sección. El conjunto contiene calificaciones de 1628 películas por 943 usuarios, cada uno de los cuales calificaron al menos 20 películas.

Cree un nuevo método llamado loadMovieLens en *recommendations.py* para cargar el dataset:

```
def loadMovieLens(path='./data/movielens'):
  # Obteniendo los titulos de las peliculas
  movies={}
  for line in open(path+'/u.item'):
    (id,title)=line.split('|')[0:2]
    movies[id]=title
  # cargando los datos
  prefs={}
  for line in open(path+'/u.data'):
    (user,movieid,rating,ts)=line.split('\t')
    prefs.setdefault(user,{})
    prefs[user][movies[movieid]]=float(rating)
  return prefs
```

En tu sesión de Python, carga los datos y revisa algunas calificaciones para cualquier usuario:

```
>> reload(recommendations)
>> prefs=recommendations.loadMovieLens( )
>> prefs['87']
{'Birdcage, The (1996)': 4.0, 'E.T. the Extra-Terrestrial (1982)': 3.0,
 'Bananas (1971)': 5.0, 'Sting, The (1973)': 5.0, 'Bad Boys (1995)': 4.0,
 'In the Line of Fire (1993)': 5.0, 'Star Trek: The Wrath of Khan (1982)': 5.0,'Speechless (1994)': 4.0,
etc...
```

Ahora puedes obtener recomendaciones basadas en el usuario:

```
>> recommendations.getRecommendations(prefs,'87')[0:30]
[(5.0, 'They Made Me a Criminal (1939)'), (5.0, 'Star Kid (1997)'),
 (5.0, 'Santa with Muscles (1996)'), (5.0, 'Saint of Fort Washington (1993)'),
 etc...]
```

Dependiendo de la velocidad de tu computador, puedes notar una demora cuando obtienes recomendaciones de esta forma. Esto es porque estas trabajando con un dataset mucho más largo. Ahora vamos a intentar hacer recomendaciones basadas en ítems.

```
>> itemsim=recommendations.calculateSimilarItems(prefs,n=50)
100 / 1664
200 / 1664
etc...
>> recommendations.getRecommendedItems(prefs,itemsim,'87')[0:30]
[(5.0, "What's Eating Gilbert Grape (1993)"), (5.0, 'Vertigo (1958)'),
 (5.0, 'Usual Suspects, The (1995)'), (5.0, 'Toy Story (1995)'),etc...]
```

Aunque construir el diccionario toma un tiempo largo, las recomendaciones son instantáneas después de que este está construido. Además el tiempo que toma obtener recomendaciones no se incrementa cuando el número de usuarios aumenta.

Este es un gran *dataset* para experimentar y ver cómo métodos diferentes de puntuación afectan a las salidas, y para entender como el filtrado basado en usuarios y en ítems se desempeñan de forma diferente. El site *GroupLens* tiene algunos otros *datasets* para jugar, incluyendo libros, bromas, y más películas.

Filtrado basado en usuarios o filtrado basado en ítems?

El filtrado basado en ítems es significativamente más rápido que el basado en usuarios al obtener una lista de recomendaciones para un *dataset* grande, pero este no tiene el trabajo adicional del mantenimiento de la tabla de similitudes. También hay una diferencia de exactitud que depende que cuan disperso es el *dataset*. En el ejemplo de las películas cada crítico ha calificado casi todas las películas, el conjunto de datos es denso.

> *Para aprender más sobre las diferencias en desempeño entre estos algoritmos, revisar el paper llamado "Item-based Collaborative Filtering Recommendation Algorithms" by Sarwar et al. at http://citeseer.ist. psu.edu/sarwar01itembased.html.*

Habiendo dicho esto, el filtrado basado en usuarios es fácil de implementar y no tiene pasos extra, de forma que es más apropiado con datasets en memoria que cambian frecuentemente. Finalmente, en algunas aplicaciones, mostrar personas con las cuales los usuarios tienen preferencias similares a ellos mismos es también valiosos – quizás no sea algo que se quiera hacer en una tienda, pero posiblemente sea apropiado en un site de recomendación de música o de links.

Has aprendido ahora cómo calcular puntajes similares y cómo usar estos para comparar gente e ítems. En el siguiente capítulo usaremos algunas de estas ideas para encontrar grupos de personas similares utilizando algoritmos de clustering.

Ejercicios:

1. Coeficiente de Tanimoto: investigar qué es el coeficiente de similitud de Tanimoto. ¿En qué casos puede ser usada como una métrica de similitud en lugar de la Distancia Euclidiana? Cree una nueva función de similitud usando el coeficiente de Tanimoto.
2. Eficiencia basada en el usuario: el algoritmo de filtrado basado en usuarios es ineficiente porque compara a un usuario con todos los otros cada vez que una recomendación se necesita. Escribir una función que precalcule

similitudes. Altere el código de las recomendaciones para usar solamente los cinco usuarios más parecidos para obtener recomendaciones.

Capítulo 3
DESCUBRIENDO GRUPOS

Este capítulo introduce el agrupamiento de datos, un método para descubrir y visualizar grupos de cosas, gente o ideas que están relacionadas. En este capítulo aprenderemos cómo prepara datos desde una variedad de fuentes con dos algoritmos de agrupamiento diferentes más una métrica de distancia, código de visualización de imágenes para visualizar los datos y finalmente un método para proyectar en dos dimensiones conjuntos muy complejos de datos.

El agrupamiento es usado frecuentemente en aplicaciones de datos intensivas. Los vendedores que hacen seguimiento de las preferencias de los usuarios pueden usar esta información para detectar grupos de clientes con similares patrones de compra automáticamente, adicionalmente a información demográfica. Personas de edades similares puede tener distintos estilos de vestir, pero con el uso de agrupación de "islas de moda" pueden ser descubieratas y usadas para desarrollar una estrategia de marketing. La agupación es también ampliamente usada en biología computacional para encontrar grupos de genes que exhiben comportamientos similares, los cuales pueden indicar que ellos responden a un tratamiento en la misma forma o son parte de alguna rama biológica.

Dado que este libro trata sobre inteligencia colectiva, los ejemplos en este capítulo toman ejemplos en los cuales muchas personas contribuyen con información diversa. El primer ejemplo que veremos es sobre blogs, los tópicos que ellos discuten y su particular uso de las palabras para mostrar que los blogs pueden ser agrupados de acuerdo a su texto y las palabras pueden ser agrupadas por su uso. El segundo ejemplo nos mostrará una un site comunitario donde la gente lista cosas que ellos tienen y cosas que les gustaría tener, y nosotros usamos esta información para mostrar cómo la gente puede ser clasificada dentro de grupos.

Aprendizaje Supervisado vs. No supervisado

Las técnicas que utilizan ejemplos de entradas y salidas para aprender cómo hacer predicciones son conocidas como métodos de aprendizaje supervisado. Exploraremos muchos métodos de aprendizaje supervisado en este libro, incluyendo redes neuronales, árboles de decisión y clasificadores bayesianos. Las aplicaciones usan estos métodos de aprendizaje examinando un conjunto de entradas y salidas esperadas. Cuando queremos extraer información usando uno de estos métodos, ingresamos un conjunto de entradas y esperamos que la aplicación produzca una salida basada en lo que ha sido aprendido antes.

El clustering es un ejemplo de aprendizaje no supervisado. A diferencia de una red neuronal o un árbol de decisión, los algoritmos de aprendizaje no supervisado no

son entrenados con ejemplos de respuestas correctas. Su propósito es encontrar estructuras dentro de un conjunto de datos donde ninguna pieza de datos es la respuesta. En el ejemplo de moda mencionado, los clusters no le dicen al vendedor lo que un vendedor quiere comprar, sino que hacen predicciones del grupo en el que la persona encaja. El objetivo de los algoritmos de clustering es tomar los datos y descubrir los distintos grupos que existen dentro de estos.

Otros ejemplos de aprendizaje no supervisado incluyen factorización de matriz no negativa, el cual será discutido luego, y mapas autoorganizados.

Vectores de Palabras

El modo normal de preparar datos para clustering es determinar un conjunto común de atributos numéricos que pueden ser usados para comparar ítems. Esto es muy dimilar a lo que vimos en el capítulo anterior, donde los rankings de las críticas fueron comparadas en un conjunto común de películas, y cuando la presencia o ausencia de bookmarks se tradujo como 0 o 1.

Categorización de los Bloggers

En este capítulo trabajaremos sobre un par de datasets de ejemplo. En el primer dataset, los ítems que se agruparán son un conjunto de 120 de los mejores blogs, y los datos que se agruparán de ellos es el numero de veces que un conjunto de palabras aparece en el feed de cada blog. Un pequeño subconjunto de esto se muestra en la Ilustración 3-1

	"china"	"kids"	"music"	"yahoo"
Gothamist	0	3	3	0
GigaOM	6	0	0	2
Quick Online Tips	0	2	2	22

Ilustración 3-1 Subconjunto de las frecuencias de palabras del blog

Mediante la agrupación de blogs basados en frecuencia de palabras, será posible determinar si hay grupos de palabras que se escriben frecuentemente sobre temas similares o escritos en estilos similares. Tales resultados pueden ser muy útiles para buscar, catalogar y descubrir la gran cantidad de blogs que están en línea actualmente.

Para generar este dataset, debes descargar los feeds de un conjunto de blogs, extraer el texto de las entradas y crear una tabla de frecuencias de palabras. Si deseas saltar estos pasos para crear un dataset, puedes utilizar el archivo blogdata.txt que viene en el código del libro.

Contando las palabras en un Feed

La mayoría de blogs puedes leerse en línea o mediante sus feeds RSS. Un RSS es un simple documento XML que contiene información sobre el blog y todas sus entradas. El primer paso para para generar un contador de palabras para cada blog es hacer *parse* de estas palabras. Afortunadamente hay un excelente módulo para hacer esto llamado Universla Feed Parser, el cual puedes descargar de http://www.feedparser.org.

Este módulo hace fácil obtener el título, enlaces y entrada de cualquier RSS o feed Atom. El siguiente paso es crear una función que extraiga todas las palabras de un feed. Cree un nuevo archivo llamado *generatefeedvector.py* e inserte el siguiente código.

```
import feedparser
import re

# Devuelve el titulo y un diccionario de conteo de palabras de un feed RSS

def getwordcounts(url):
  # Analiza lexicamente el feed
  d=feedparser.parse(url)
  wc={}

  # itera sobre todas las entradas
  for e in d.entries:
    if 'summary' in e: summary=e.summary
    else: summary=e.description

    # extrae una lista de palabras
    words=getwords(e.title+' '+summary)
    for word in words:
      wc.setdefault(word,0)
      wc[word]+=1
  return d.feed.title,wc
```

Los feeds Atom y RSS siempre tienen un título y una lista de entradas. Las cuales usualmente tienen una etiqueta summary o description que contiene el texto de las entradas. La función getwordcounts pasa este resumen a getwords, la cual quita todos el HTML y divide las palabras con caracteres no alfabéticos, retornándolos en una lista. Agregue getwords a *generatefeedvector.py*:

```
def getwords(html):
  # quita todas las etiquetas HTML
  txt=re.compile(r'<[^>]+>').sub('',html)

  # Divide las palabras con caracteres no alfabeticos
  words=re.compile(r'[^A-Z^a-z]+').split(txt)
  # Convierte a minusculas
  low = [word.lower() for word in words if word!='']
```

```
  return low
```

Ahora necesitas una lista de feeds para trabajar. Si gustas puedes generar una lista de URLs para un conjunto de blogs tú mismo, o puedes utilizar una lista prefabricada de 100 RULs de RSS. Esta lista fue creada tomando los feeds de los blogs más recomendados y removiendo estos que no contienen el texto completo o donde la mayor parte son imágenes. Puedes descargar la lista de feedlist.txt.

Este es un texto plano con una URL en cada línea. Si tienes tu propio blog o algunos favoritos y quieres ver cómo estos se comparan a algunos de los blogs más populares aparte de estos, puedes agregarlos a este archivo.

El código para iterar sobre los feeds y generar el dataset será el código principal en generatefeedvector.py (que no es una función). La primera parte del código itera sobre cada una de las líneas en feedlist.txt y genera el conteo de palabras para cada blog, de forma que el número de blogs en los que cada palabra aparecer en (apcount). Agrege este código al final de generatefeedvector.py:

```
import codecs
charset = "utf-8"

apcount={}
wordcounts={}
file=open('feedlistmia.txt')
feedlist=[line for line in file]
for feedurl in feedlist:
  try:
    print(feedurl)
    title,wc=getwordcounts(feedurl)
    wordcounts[title]=wc
    for word,count in wc.items():
      apcount.setdefault(word,0)
      if count>1:
        apcount[word]+=1
  except:
    print ('Failed to parse feed %s' % feedurl)
```

El siguiente paso es generar una lista de palabras que serán usadas actualmente usadas en el conteo para cada blog. Dado que las palabras como “the” aparecen en la mayoría de ellos, y otras como “flim-fllam” pueden solo aparecer una vez, puedes reducir el número total de palabras incluyendo la selección solo aquellas palabras que están dentro de los porcentajes máximos y mínimos. En este caso puedes empezar con 10% como límite inferior y 50% como límite superior, pero es necesario experimentar con estos números si encuentras apareciendo muchas palabras comunes o muchas palabras extrañas:

```
wordlist=[]
for w,bc in apcount.items():
```

```
  frac=float(bc)/len(feedlist)
  if frac>0.1 and frac<0.5:
    wordlist.append(w)
```

El pasó final para usar la lista de palabras y la lista de blogs para crear un archivo de texto conteniendo una gran matriz de todas las palabras contadas para cada uno de los blogs:

```
out=codecs.open('blogdata.txt','w',charset)
out.write('Blog')
for word in wordlist: out.write('\t%s' % word)
out.write('\n')
for blog,wc in wordcounts.items():
  print (blog)
  out.write(blog)
  for word in wordlist:
    if word in wc: out.write('\t%d' % wc[word])
    else: out.write('\t0')
  out.write('\n')
```

Para generar el archivo de conteo de palabras, corre generateffedvector.py desde la línea de comandos.

```
c:\code\blogcluster>python generatefeedvector.py
```

Descargar todos estos feeds puede tomar varios minutos, pero esto eventualmente genera un archivo de salida llamado blogdata.txt. abra este archivo para verificar que contiene una tabla separada por tabulaciones con columnas de palabras y filas de blogs. Este formato de archivo puede ser utilizado por las funciones en este capítulo, de forma que después puedes crear un *dataset* diferente o incluso guardar una hoja apropiadamente formateada como un texto separado por tabulaciones sobre la cual se usaran estos algoritmos.

Cluster jerárquico

El clustering construye una jerarquía de grupos mezclando continuamente los dos grupos más similares. Cada uno de estos grupos empieza con un solo ítem, en este caso un blog individual. En cada iteración este método calcula la distancia entre cada par de grupos, y los más cercanos son combinados para formar un grupo nuevo. Esto se repite hasta que hay solo un grupo.

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

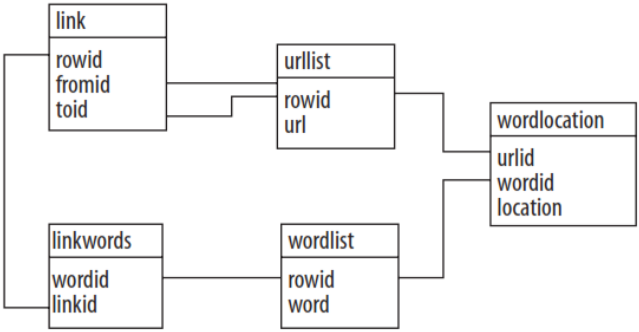

Ilustración 4-1 Esquema para el motor de búsqueda

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 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(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.

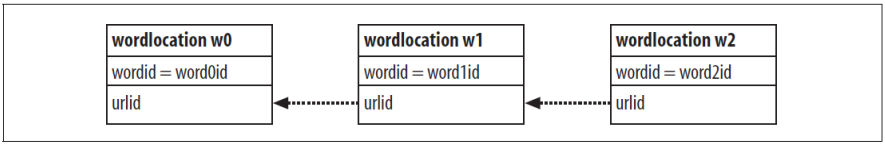

Ilustración 4-2 Tabla unida por getmachrows

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).