Proyecto de prácticas: Recuperador de Noticias

Sistemas de Almacenamiento y Recuperación de Información

Roberto Pérez Rico
Carlos Fernández Jordán
Zhuqing Wang
Alonso Culebras Tévar

1. Introducción

A continuación le hablaremos sobre el proyecto de prácticas de la asignatura de Sistemas de Almacenamiento y Recuperación de Información del Recuperador de Noticias.

2. Funcionalidades básicas

Indexar una noticia:

Para el caso básico, hemos cargado el archivo de noticias json y, a partir de éste hemos cogido cada noticia de cada uno de los archivos y la hemos tokenizado (elimina símbolos no alfanuméricos y dividiéndola por espacios). A continuación, por cada una de las palabras de cada noticia hemos creado un índice para ella, en caso de que no exista y, en caso contrario, añadirla con el id de la noticia a la que pertenece. Cabe distinguir que este es el proceso de indexación de que el campo 0 de field sea un artículo, para el caso contrario se crearía un nuevo token de artículo.

```
with open(filename) as fh:
    jlist = json.load(fh)

if self.positional:
    self.index_file_positional(filename)
else:
    self.docs[str(self.docId)] = filename
    for new in jlist:
        for field in self.fields:
            if field[1]:
                a = new[field[0]]
                article = self.tokenize(a)
                for word in article:
                    if self.index[field[0]].get(word) is None:
                        self.index[field[0]][word] = [self.newsId]
                    else:
                        if self.newsId not in self.index[field[0]][word]:
                            self.index[field[0]][word].append(self.newsId)

            else:
                token = new[field[0]]
                if self.index[field[0]].get(token) is None:
                    self.index[field[0]][token] = []
                if (self.docId, self.newsId) not in self.index[field[0]][token]:
                    self.index[field[0]][token].append((self.docId, self.newsId))

        self.news[self.newsId] = (self.docId, id)
        self.newsId += 1
    self.docId += 1
```

Recuperador de noticias

A la hora de recuperar noticias hemos implementado el método solve_query que se encarga de realizar el parsing de consulta. Para el caso de la implementación básica:

```
335
336         if query is None or len(query) == 0:
337             return []
338
339         # sacar postiing list de cada palabra y hacer su query (and or or)
340
341         query = str(query).split(' ')
342         i = 0
343         res = []
344         while len(query) > i:
345             if query[i] == 'NOT':  # FIRST TERM
346                 res = self.reverse_posting(self.get_posting(query[i + 1]))
347                 i += 2
348             elif query[i] == 'AND':
349                 if query[i + 1] == 'NOT':
350                     res = self.and_posting(res,
    self.reverse_posting(self.get_posting(query[i + 2])))
351                     i += 3
352                 else:
353                     res = self.and_posting(res, self.get_posting(query[i + 1]))
354                     i += 2
355             elif query[i] == 'OR':
356                 if query[i + 1] == 'NOT':
357                     res = self.or_posting(res,
    self.reverse_posting(self.get_posting(query[i + 2])))
358                     i += 3
359                 else:
360                     res = self.or_posting(res, self.get_posting(query[i + 1]))
361                     i += 2
362             else:
363                 res = self.get_posting(query[i])
364                 i += 1
365         return res
```

AND posting

```
def and_posting(self, p1, p2):
    res = []
    i = 0
    j = 0
    p1.sort()
    p2.sort()
    while i < len(p1) and j < len(p2):
        if p1[i] == p2[j]:
            res.append(p1[i])
            i += 1
            j += 1
        elif p1[i] < p2[j]:
            i += 1
        else:
            j += 1
    return res
```

OR posting

```
def or_posting(self, p1, p2):
    res = []
    i = 0
    j = 0
    p1.sort()
    p2.sort()
    while i < len(p1) and j < len(p2):
        if p1[i] == p2[j]:
            res.append(p1[i])
            i += 1
            j += 1
        elif p1[i] < p2[j]:
            res.append(p1[i])
            i += 1
        else:
            res.append(p2[j])
            j += 1

    while i < len(p1):
        res.append(p1[i])
        i += 1

    while j < len(p2):
        res.append(p2[j])
        j += 1

    return res
```

NOT posting

```
def reverse_posting(self, p):

    return [res for res in list(self.news.keys()) if res not in p]
```

3. Funcionalidades extras

3.1. Stemming

El *stemming* de las noticias que hemos hecho está formado por un diccionario *self.sindex* de 2 niveles de profundidad *[field] : [stem]* y los valores de *[stem]* es una lista de tokens del stem.

En el método *make_stemming* lo que hace es simplemente recorrer las índices invertidos e ir añadiendo al índice de stems los términos que corresponden:

```
if self.multifield:
    multifield = ['title', 'date', 'keywords', 'article', 'summary']
else:
    multifield = ['article']
# Se aplica stemming a cada token del self.index[field] y se añade al indice de stems
for field in multifield:
    for token in self.index[field].keys():
        tokenAux = self.stemmer.stem(token)
        if self.sindex[field].get(tokenAux) is None:
            self.sindex[field][tokenAux] = [token]
        else:
            if token not in self.sindex[field][tokenAux]:
                self.sindex[field][tokenAux] += [token]
```

En cuanto al método get_stemming, nos permite obtener la posting list de los stems y lo que hace es recorrer los términos a los que contenga la entrada de un stem en el índice y sacar la unión de las posting lists mediante el uso del método or_posting.

```
# se obtiene el stem de un termino
stem = self.stemmer.stem(term)
res = []

# Se hace la unión de las posting list de cada termino que contenga la entrada en el indice de stems
if stem in self.sindex[field]:
    for token in self.sindex[field][stem]:
        res = self.or_posting(res, list(self.index[field][token].keys()))
        print(self.index[field][token])
return res
```

3.2. Permuterm

Para crear un índice permuter hemos necesitado de los métodos get_permuterm, make_permuterm y el índice invertido permuterm. Se pueden lanzar dos tipos de wildcard: una usando '*', que representa cualquier otro carácter o cadena de caracteres y otra usando?' que representa un único carácter entre los caracteres donde ha sido introducido.

Por ejemplo, la query 'c*sa' devolvería palabras como casa, cosa... entre otras.

Como segundo ejemplo, la query 'bue?' devolvería palabras como buey, buen... entre otras.

Si se usa la flag -P en el indexador, se podrán hacer querys más adelante en el Searcher, de lo contrario no se podrán crear este tipo de queries.

La estrategia que se ha llevado a cabo para encontrar los índices permuterm ha sido la siguiente:

1. Obtenemos la query permuterm a usar sobre nuestro índice invertido permuterm.
2. Distinguimos de que tipo de query se trata, si '*' o '?'.
3. Si se trata de '*', añadimos a nuestra lista de posting list las de los términos que empiecen por x* y que acaben por *y, donde x sería el principio de la palabra hasta la wildcard y 'y' sería desde la wildcard hasta el final del término.
4. Si se trata de '?', añadimos a nuestra lista de posting list las de los términos que como anteriormente empiecen y acaben por (x*y) y que además la longitud del término sea igual a la longitud de x + la longitud de y + 1.

3.3. Ranking

Para establecer un ranking para las noticias se ha elegido computar un índice de jaccard para cada noticia con respecto a su query. Se ha elegido este método ya que premia los artículos que son más concisos y su computación no es nada costosa en comparación a otras distancias vistas en teoría como por ejemplo la distancia coseno.

Antes de realizar el cómputo de jaccard, se eliminan de la query todos los elementos que no sean términos que se buscan en la query. Los 'NOT' se reemplazan por la cadena

‘nnoott’. Esto se hace para poder eliminar los términos siguientes a un NOT, ya que no queremos encontrar estos términos en nuestros documentos, y evitar confundirlo con un término real de la query. Tras esto se tokeniza la cadena resultante y se eliminan los ‘nnoott’ y los términos que no nos interesan.

```
# Se eliminan los elementos de la query que no son términos. Puesto que se eliminará los términos después de un NOT,
# este se reemplaza por nnoott para evitar que se pueda confundir con un término tras pasarlo a minúsculas.
query = query.replace('AND', '')
query = query.replace('OR', '')
query = query.replace('NOT', 'nnoott')
query = self.tokenize(query)
i = 0
while i<len(query):
    if query[i]=='nnoott':
        del query[i:i+2]
        i+=1
    i+=1
```

Para cada documento que se haya devuelto tras la consulta computamos su índice de jaccard con respecto a la query.

```
# Se calcula la distancia de jaccard para cada noticia devuelta por la query
for elem in result:
    with open(self.docs.get(str(self.news[elem][0]))) as fh:
        document = json.load(fh)
        document = document[0]['article']
    # Se guardan los resultados en una tupla formada por el identificador
    #  la noticia y su ranking con respecto a la query.
    rank.append([self.jaccard(query, self.tokenize(document)), elem])
```

Esto se consigue calculando la intersección y unión de los conjuntos resultantes y calculando el cociente entre la longitud de ambas. El valor resultante irá de 1 a 0, siendo 1 la mejor puntuación y 0 la peor.

```
Computa la metrica de jaccard el documento enviado.
J(A,B) = |A ∧ B| / |A ∨ B|
'''

intr = [value for value in query if value in article]
union = article + [value for value in query if value not in intr]
# DEvolvemos el cociente entre las longitudes de la intersección y la union de los conjuntos.
return len(intr) / len(union)
```

Al ser la longitud de la unión el divisor en esta operación, cuanto más grande sea el documento con el que nos encontramos, más grande será la unión y por tanto menor el resultado de la división. De esta forma se premian los documentos que contienen los términos que buscamos y son de menor longitud, ya que cuanto más corto és un artículo más probable será que contenga la respuesta a nuestra consulta mientras que uno de gran longitud puede usar todos los términos que buscamos pero en diferentes contextos y no contestar a nuestra consulta a pesar de cumplir las restricciones que impone la query.

Se devuelve una lista cuyos elementos son los identificadores de los documentos ordenados con respecto a el ranking y el valor de este.

```
rank.sort(key=lambda tup: tup[0], reverse=True)
return rank
```

3.4. Búsquedas posicionales

Esta parte de la ampliación no hemos conseguido hacerla funcionar, pero aún así explicaremos las conclusiones que hemos extraído y como hemos intentado implementarla. Parte del código de dicha parte ha sido eliminado en la versión final ya que no era funcional.

El caso de las búsquedas posicionales requerimos de hacer unas adaptaciones en el código de `index_file`, ya que deberemos guardar para cada palabra de la noticia, su posición en la misma.

En el método `index_file`, crearemos la llamada a un método nuevo llamado `index_file_positional` para el caso de búsquedas posicionales:

```
if self.positional:
    self.index_file_positional(filename)
else:
```

(continúa el código de index_file)

El método `index_file_positional` es muy similar al método original a diferencia de la variable idWord que se le pasaría a la propia palabra en el índice y llevaría el número de palabra con respecto al documento.

```
idWord = 0
for word in article:
    if self.index.get(word) is None:
        self.index[word] = [(self.newsId, idWord)]
    else:
        self.index[word].append((self.newsId, idWord))
    idWord += 1
self.news[self.newsId] = (self.docId, id)
self.newsId += 1
id += 1
```

Para esta implementación, también ha hecho falta modificar el método `solve_query` para que detecte las cadenas a consultar. Por ejemplo, la cadena ¨fin de semana¨ se trata de una cadena que al eliminar los símbolos alfanuméricos quedaría como term1 term2 term3 para los cuáles debemos crear el caso en el propio método:

```
else:
    j= 0
    terms = []
    while len(query)<(i + j) and query[i + j] != 'NOT' and query[i + j] != 'AND' and query[i + j] != 'OR':
        terms.append(q[i + aux])
        j+=1
    if terms == 1:
        res = self.get_posting(query[i])
        i += 1
    else:
        res = res.append(self.get_positionals(terms))
```

En esta parte del código almacenaremos en terms los términos de la consulta que no estén seguidos de ´NOT´, ´AND´ o ´OR´, por lo que deberán estar seguidos de otro término. En el caso de que no haya más que un término en terms, se trataría del último término de la

consulta. En el caso contrario, se trata de términos contiguos y, por consiguiente, de una búsqueda posicional. Esta búsqueda adjuntará a la respuesta (res) una llamada al método get_positionals con el atributo terms.

```
def get_positionals(self, terms, field='article'):


  sols = self.index[terms[0]]
  i = 1
  while i < len(terms):
      newsols = self.index[terms[i]]
      auxsols = []
      for sol in sols:
          for newsol in newsols:
              if (sol[0] == newsol[0] and sol[1] + 1 == newsol[1]):
                  auxsols.append(newsol)

      sols = auxsols
  return sols
```

Este sería un ejemplo de implementación del `get_positionals`. En esta implementación recorreremos la lista de términos (terms) y comprobaremos para cada uno si su id de noticia coincide con el del término sucesor y si el número de la palabra en la noticia del siguiente término es contiguo al suyo. Por último, añadiremos a soluciones (sols) los términos que coincidan con dicho criterio y retornaremos sols.

Como ya he comentado al principio, esta parte de la ampliación no funciona e incluso algún método (`index_file_positional`) no está implementado o se han borrado partes del código, pero nos gustaría que considerase esta parte de la memoria para puntuar igualmente.

4. Reparto de trabajo

Funcionalidades básicas hechas entre todos.

Stemming: Zhuqing Wang

Permuterm y solve query: Roberto Pérez Rico

Ranking: Carlos Férnandez

Posicionales: Alonso Culebras Tévar