Proyecto: Recuperador de Noticias

Sistemas de Almacenamiento y Recuperación de la Información (3CO21)
Antoni Mestre Gascón - anmesgas@inf.upv.es
Alejandro Granados Bañuls - algrabau@inf.upv.es
Mario Campos Mocholí - macammoc@inf.upv.es

1 Introducción

Como primera parte del proyecto conjunto de las asignaturas SAR y ALT, se ha realizado la implementación de un indexador de noticias y un recuperador de estas mediante consultas en lenguaje natural con conectores lógicos y funcionalidades avanzadas.

El informe está divido en diversas secciones donde se explica la implementación de las funcionalidades básicas, la de las funcionalidades extras y la explicación de la metodología de trabajo y el repartimiento de tareas.

Pese que para obtener la máxima nota es necesario implementar solo cuatro funcionalidades extras, se ha intentado implementar todas ya que consideramos que son de interés para afianzar conocimientos teóricos de la asignatura de forma práctica. Todas funcionan correctamente excepto la posicional, pero aun así se ha añadido explicando que teniamos realizar.

2 Funcionalidades básicas

En la presente sección se detallan las implementaciones de las funcionalidades básicas del proyecto: el indexado de noticias, la recuperación de noticias y la visualización de los resultados.

La implementación de las funcionalidades extras modifica drásticamente el código de las implementaciones básicas. Es por ello que, pese a explicar posteriormente estas funcionalidades no obligatorias, el código mostrado y funcionamiento en esta sección contiene trazas de las extras ya que el código final es indisoluble uno del otro.

2.1 Indexador de noticias

Paso previo a la consulta sobre un conjunto de documentos es necesario indexarlos. Como estructura de datos del índice invertido se ha decidido utilizar un diccionario con tres niveles de profundidad [campo] $\rightarrow$ [término] $\rightarrow$ [noticia].

Cada término tiene como valor una lista de diccionarios de las noticias donde aparece. Cada noticia de este diccionario tiene a su vez una lista de las posiciones del término en el documento o el número de ocurrencias en el documento, dependiendo si las búsquedas posicionales están activadas o no.

```
self.index = {'title': {},
              'date': {},
              'keywords': {},
              'article': {},
              'summary': {}
              }
```

El método *index_dir*() se ha modificado ligeramente para realizar los indexados de las funcionalidades extras si procede, para ahorrar tiempo en caso de un indexado simple.

```
def index_dir(self, root, **args):
        #[...]
        # Si se activa la función de stemming
        if self.stemming:
            self.make_stemming()
        # Si se activa la función de permuterm
        if self.permuterm:
            self.make_permuterm()
```

En cambio, el método *index_file*() es donde se ha implementado toda la lógica encargada de: generar el índice de términos, el índice de noticias y el índice de documentos.

A cada documento se le asigna un identificador secuencial de 0 a (número de documentos - 1) y se añade una entrada al diccionario de documentos guardando la ruta absoluta como valor.

A cada noticia también se le asigna un identificador secuencial de 0 a (número de noticias total - 1) y se añade una entrada al diccionario de noticias guardando como valor una tupla de (identificador documento que la contiene, posición de la noticia en el documento)

El proceso de indexado es trivial. Se recorre secuencialmente los documentos y las noticias que contienen, tokenizando estas y añadiendo cada término en el indice inverso si no estaba o actualizando el valor si ya se encontraba.

```
def index_file(self, filename):
    with open(filename) as fh:
        jlist = json.load(fh)
        self.docs[self.doc_cont] = filename

        # Contador de la posición de una noticia en un fichero
        contador_noticia = 0
        for noticia in jlist:
            self.news[self.new_cont] = [self.doc_cont,
                                        contador_noticia]
        #[...]

        for token in contenido:
```

```
            # Si el token no esta en el diccionario de tokens
            if token not in self.index[field]:
                if self.positional:
                    self.index[field][token] = {
                        self.new_cont: 1}
                else:
                    self.index[field][token] = {
                        self.new_cont: [posicion_token]}
            # Si el token esta ya...
            else:
                # ...si no existe la noticia en el token
                if self.new_cont not in self.index[field][token]:
                    if self.positional:
                        self.index[field][token][self.new_cont] = 1
                    else:
                        self.index[field][token][self.new_cont] = [
                            posicion_token]
                else:
                    # Si no...
                    if self.positional:
                        self.index[field][token][self.new_cont] += 1
                    else:
                        self.index[field][token][self.new_cont]
                            += [posicion_token]

            posicion_token += 1
        self.new_cont += 1
        contador_noticia += 1
    self.doc_cont += 1
```

Finalmente el método *show_stats*() emula el comportamiento del ejemplo del boletín. Simplemente se accede a los índices correspondientes y se realizan las estadísticas demandadas, según los argumentos.

2.2 Recuperador de noticias

Una vez indexada una colección de documentos, es necesario realizar la lógica para realizar consultas sobre estos.

El método *solve_query*() ha sido el más difícil de implementar por las diversas modificaciones que ha habido que realizar tras implementar cada funcionalidad extra y conseguir que todas funcionaran conjuntamente. En el código se han dejado dos versiones, una primera comentada más detallada pero enrevesada, y una segunda más concisa y eficiente y que es la que se ha utilizado.

Para desarrollarlo, se ha analizado las consultas propuestas y a partir de estas se ha deducido las prioridades en la aplicación de cada funcionalidad obteniendo: paréntesis → multicampo → wildcard queries → posicionales → conectores básicos.

El algoritmo funciona de forma recursiva para los paréntesis (subconsultas) y de forma iterativa para el resto de funcionalidades. Se ha optado por implementarlo de esta forma por simplicidad. En un primer lugar se preprocesa la consulta para poder separar los términos y símbolos:

```
# Preprocesamiento de la consulta
query = query.replace('"', '')
query = query.replace('(', ' ( ')
query = query.replace(')', ' ) ')
q = query.split()
```

Seguidamente se entra en un bucle que recorre la consulta procesada (que es una lista ahora) realizando las funcionalidades en el orden descrito anteriormente. Cada funcionalidad obtiene como resultado una posting list, el como se detallara en la sección correspondiente.

Al final de este bucle se codifican los conectores básicos y finalmente se entra en otro bucle para resolverlos.

El método *get_posting*() obtiene la posting list de un término teniendo en cuenta: si es un término con comodines, esta activada la función de stemming o es un término normal.

En los dos primeros casos se llaman a otras funciones auxiliares. En el tercero, se accede directamente al índice invertido de términos y si existe en este, se devuelve su posting list.

```
def get_posting(self, term, field='article', wildcard='False'):
    res = []
    # Posting list de una wildcard query
    if '*' in term or '?' in term:
        res = self.get_permuterm(term, field)
    # Posting list de un stem
    elif self.use_stemming and not wildcard:
        res = self.get_stemming(term, field)
    # Posting list de un termino
    else:
        if term in self.index[field]:
            res = list(self.index[field][term].keys())
    return res
```

Finalmente se han implementado los conectores básicos NOT, AND y OR. El NAND propuesto no nos era útil y se ha optado por no implementarlo.

Para realizar el conector NOT se obtiene la lista con todos los documentos y se elimina de esta los documentos que aparecen en la posting list del parametro:

```
def reverse_posting(self, p):
    # Obtenemos lista de todas las noticias
    res = list(self.news.keys())
```

```
    # Recorremos la posting list
    for post in p:
        # Eliminamos la noticia de la lista de todas las noticias
        res.remove(post)
    return res
```

El conector AND ha sido una simple traducción del pseudocódigo visto en teoría a Python. Por la naturaleza de la implementación del indexador, las posting list ya estan ordenadas:

```
def and_posting(self, p1, p2):
    res = []
    i = 0
    j = 0
    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
        elif p1[i] >= p2[j]:
            j += 1
    return res
```

El conector OR también ha sido una simple traducción del pseudocódigo visto en teoría a Python:

```
def or_posting(self, p1, p2):
    res = []
    i = 0
    j = 0
    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
        elif p1[i] >= p2[j]:
            res.append(p2[j])
            j += 1
    for pos in range(i, len(p1)):
        res.append(p1[pos])

    for pos in range(j, len(p2)):
```

```
            res.append(p2[pos])
    return res
```

2.3 Visualización resultados

Como resultado de una consulta es necesario mostrar la información de la respuesta.

El método *solve_and_count*() no se ha modificado mientras que *solve_and_show*() se ha implementado para emular el comportamiento del resultado del boletín según los parámetros.

El método *snippet*() ha sido interesante de realizar ya que nos ha hecho darnos cuenta de lo difícil que es mostrar un buen fragmento representante de la noticia según la consulta. Primero se preprocesa la consulta para intentar eliminar términos y símbolos (como comodines o conectores lógicos) que no nos interesan:

```
# Se preprocesa la consulta
query = query.replace('"', '')
query = query.replace('*', '')
query = query.replace('(', '')
query = query.replace(')', '')
query = query.replace('?', '')
# Junta el 'NOT' a una palabra, creando una inexistente
query = query.replace('NOT ', 'NOT')
# Palabra rara para tener en cuenta campo multifield
query = query.replace(':', 'HZMPOSICIONAL')
query = self.tokenize(query)
```

También tenemos en cuenta si una palabra es de búsqueda multicampo para mostrar resultados de ese campo en concreto y no solo del campo por defecto 'article'.

Después del preprocesamiento, por cada término de la consulta, se intenta encontrar en el campo correspondiente añadiendo un fragmento de las palabras anteriores y posteriores a la primera ocurrencia de dicho término. Se tiene en cuenta si el término está más al principio o al final.

Es posible que este método devuelva un snippet vacío, sobretodo con consultas con comodines, pero como en el ejemplo aparecen snippets vacios hemos asumido que era una opción factible.

3 Funcionalidades extras

3.1 Stemming

El stemming Para crear el índice de stems se ha optado por un diccionario de dos niveles de profundidad [campo] → [stem], donde el valor es una lista de términos al que hace referencia un stem.

Simplemente se recorre el índice invertido de términos, se obtiene el stem de cada uno y se va añadiendo al índice de stems, de forma similar al indexado normal pero sin guardar la posición ni las ocurrencias:

```
def make_stemming(self):
    # Si se activa la función multifield
    if self.multifield:
        multifield = ['title', 'date', 'keywords',
                      'article', 'summary']
    else:
        multifield = ['article']
    for field in multifield:
        # Se aplica stemming a cada token del self.index[field]
        # En este caso solo se guarda la noticia, no la posición
        for token in self.index[field].keys():
            token_s = self.stemmer.stem(token)
            if token_s not in self.sindex[field]:
                self.sindex[field][token_s] = [token]
            else:
                if token not in self.sindex[field][token_s]:
                    self.sindex[field][token_s] += [token]
```

Para obtener la posting list de un stem se llama a *get_stemming*() el cual recorre los términos a los que apunte la entrada de un stem en el indice y realiza el OR lógico implementado para unir las posting lists:

```
def get_stemming(self, term, field='article'):
    # Se obtiene el stem de un término
    stem = self.stemmer.stem(term)
    res = []
    # Se hace la unión de las posting list
    if stem in self.sindex[field]:
        for token in self.sindex[field][stem]:
            # Se utiliza el OR propio por eficiencia
            res = self.or_posting(
                res, list(self.index[field][token].keys()))
    return res
```

Finalmente se han modificado todos los demás métodos según se necesitaba hacer stemming o no.

3.2 Multifield

A priori nos parecía una de las funcionalidades más complejas de implementar pero no ha sido así.

Añadiendo el nivel de profundidad de [field] a los índices se ha conseguido implementar sencillamente esta funcionalidad.

Posteriormente se han modificado los métodos para recorrer el nuevo nivel según sea una consulta con multicampo o no y modificado la declaración de los métodos por si hacia falta pasarle un argumento extra que indicara el campo.

Para realizar las búsquedas multicampo se ha modificado el *solve_query*() simplemente añadiendo:

```
if ':' in term:
    field = term[0:term.find(':')]
    term = term[term.find(':') + 1:]
else:
    field = 'article'
```

3.3 Búsquedas posicionales

No hemos conseguido implementar correctamente esta funcionalidad extra. La parte de indexación del indice de permuterms si que funciona correctamente pero la recuperación no.

Como se ha indicado anteriormente, si esta opción está activada, el índice de términos guarda una lista ordenada con la posición de cada término en un documento.

El método *solve_query*() se ha modificado añadiendo el siguiente fragmento encargado de comprobar si es una secuencia de términos y por tanto corresponde a una búsqueda posicional:

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

El método *get_positional*() obtiene la posting list comprobando que, para cada termino de la búsqueda posicional, existe una entrada para el siguiente término con la posición siguiente.

3.4 Ranking

En un primer momento se había decidido implementar un ranking por similitud coseno de un par consulta y documento, pero debido al gran número de cálculos

que debía realizar y el ruido que contienen las consultas y los documentos (stopwords), finalmente se ha optado por un ranking por coeficiente de Jaccard.

El método que calcula la métrica tiene en cuenta si se permiten búsquedas multicampo para calcular su valor conjunto. Simplemente realiza el coeficiente de la intersección de la consulta y documento entre su unión.

```
def jaccard(self, query, documento):
    query = set(self.tokenize(query))
    metrica_total = 0
    # Si la función de multifield está activada
    if self.multifield:
        for field in ['title', 'article', 'date',
                      'keywords', 'summary']:
            if field != 'date':
                documento_aux = set(self.tokenize(documento[field]))
            else:
                documento_aux = set([documento[field]])

            metrica_total += len(query.intersection(documento_aux)
                                 ) / len(query.union(documento_aux))
    else:
        documento = set(self.tokenize(documento['article']))
        metrica_total = len(query.intersection(
            documento)) / len(query.union(documento))
    return round(metrica_total, 6)
```

Finalmente el método *rank_result*() obtiene la métrica de cada documento de una respuesta de una consulta y la ordena.

3.5 Permuterm

Para crear el índice de permuterms se ha optado por un diccionario de dos niveles de profundidad [campo] → [permuterm], donde la clave es el permuterm y el valor la lista de términos al que hace referencia.

Simplemente se recorre el índice de términos, se realiza una lista de permuterms de un término y se añade al índice de permuterms, de forma similar al indexado normal de términos pero sin guardar la posición ni las ocurrencias:

```
def make_permuterm(self):
    # Si se activa la función multifield
    if self.multifield:
        multifield = ['title', 'date', 'keywords',
                      'article', 'summary']
    else:
        multifield = ['article']
    for field in multifield:
```

```
        # Se crea la lista de permuterms de un token
        # En este caso solo se guarda la noticia, no la posición
        for token in self.index[field]:
            token_p = token + '$'
            permuterm = []
            for _ in range(len(token_p)):
                token_p = token_p[1:] + token_p[0]
                permuterm += [token_p]

            for permut in permuterm:
                if permut not in self.ptindex[field]:
                    self.ptindex[field][permut] = [token]
                else:
                    if token not in self.ptindex[field][permut]:
                        self.ptindex[field][permut] += [token]
```

Para obtener la posting list de un permuterm se llama a *get_permuterm*() el cual resuelve una wildcard query dependiendo de si el comodín es '*' o '?'. Para ello recorre la wilcard query recuperando los términos que empiezan en el índice de permuterms por este. El el caso del comodín '?' se comprueba también que la longitud de la cadena sea igual.

```
def get_permuterm(self, term, field='article'):
    res = []
    # Se construye la wildcard query del termino comodín
    term += '$'
    while term[-1] != '*' and term[-1] != '?':
        term = term[1:] + term[0]
    simbolo = term[-1]
    term = term[:-1]

    for permuterm in (x for x in list(self.ptindex[field].keys())
            if x.startswith(term) and
            (simbolo == '*' or len(x) == len(term) + 1)):
        for token in self.ptindex[field][permuterm]:
            # Se utiliza el OR propio por eficiencia
            res = self.or_posting(res, self.get_posting(
                token, field, wildcard=True))

    return res
```

Finalmente se han modificado todos los demás métodos según necesitaran o no hacer una búsqueda con comodín.

3.6 Paréntesis

Este ha sido, sin lugar a dudas, la funcionalidad que más problemas no has dado al implementar porque nos ha hecho rehacer por completo varias veces el

método *solve_query*() para tratar subconsultas anidadas y comprobar que se esta haciendo la correspondiente.

Se ha modificado el método de la siguiente forma para realizar las consultas con paréntesis como subconsultas. La modificación busca la más profunda y la resuelve, y de aquí va resolviendo las demas más exteriores hasta que no queden más:

```
term = q[i]
if term == '(':
    i += 1
    q2 = ''
    aux = 0
    while aux >= 0:
        if q[i] == '(':
            aux += 1
        if q[i] == ')':
            aux -= 1
        q2 += q[i] + ' '
        i += 1
    q2 = q2.strip()
    q2 = q2[0:len(q2) - 1]
    res.append(self.solve_query(q2))
```

3.7 Funcionamiento conjunto de las funcionalidades extra

Como se ha dicho anteriormente, todas las funcionalidades extras y básicas funcionan correctamente entre sí.

4 Reparto y metodología de trabajo

Para la realización del trabajo de forma distribuida se ha utilizado un repositorio en GitHub y mediante Visual Studio Code y Git se han repartido las tareas entre los integrantes del grupo.

La coordinación de los integrantes ha sido primordial y cada uno ha influido en mayor o menor medida en cada funcionalidad, pero el reparto inicial de las funcionalidades básicas había sido de la parte 1 para Alejandro, la parte 2.1 para Mario y la parte 2.2 para Antoni.

Con el desarrollo de las funcionalidades extras el trabajo ha sido mayoritariamente conjunto, implementado la funcionalidad y modificando cada uno su parte (si hacia falta) para adaptarla a la nueva funcionalidad.