Sistema de Recuperacion de Informacion

SysRI

Perez, Jaime y Nunez, Jessica

Facultad de Matematica y Computacion de la

Universidad de la Habana

Introducción

La recuperación de información es el conjunto de actividades orientadas a facilitar la localización de determinados datos u objetos, y las interrelaciones que estos tienen a su vez con otros. Existen varias disciplinas vinculadas a esta actividad como la lingüística, la documentación o la informática. Dado la limitada capacidad de los ordenadores, originariamente, la recuperación tenía que estar limitada a unos pocos atributos o metadatos del objeto. Entre los que destacaban el autor, el título o las palabras más significativas del contenido expresado en el texto o descriptores. La asignación de estos descriptores, denominada indización, era manual.

Estos mismos metadatos son empleados actualmente en la Web Semántica por su mayor simplicidad que el lenguaje natural, facilitando la interoperabilidad y la navegación en la Web.

La indización automática trata de automatizar la asignación de términos relevantes a un documento de forma automática. La relevancia es calculada mediante cálculos estadísticos y localización del mismo. Ejemplos son tf-IDF, la eliminación de palabras vacías, el mayor valor de los términos en los títulos, en formato destacado (p.e. negrilla), etc. Muchos de estos factores son utilizados para ordenar los resultados en los motores de recuperación

Datos Utilizados

Para las pruebas realizas a la *API* utilizamos una de las colecciones de documentos ofrecidas en la orientación *20Newsgroup*

Lenguaje

El código fuente esta desarrollado en *Python* aprovechando las ventajas ofrecidas por sus bibliotecas como ***NLTK*** para el manejo de la lingüística y ***Django*** para la interfaz visual

Modelo Vectorial

Se conoce como modelo de espacio vectorial a un modelo algebraico utilizado para filtrado, recuperación, indexado y cálculo de relevancia de información. Representa documentos en lenguaje natural de una manera formal mediante el uso de vectores (de identificadores, por ejemplo, términos de búsqueda) en un espacio lineal multidimensional. Fue usado por primera vez por el sistema SMART de recuperación de información.

Muchas de las tareas de recuperación de información como la búsqueda, agrupamiento o categorización de textos tienen como primer objetivo procesar documentos en lenguaje natural. El problema que surge es que los algoritmos que pretenden resolver estas tareas necesitan representaciones internas explícitas de los documentos.

En esta representación vectorial de documentos el éxito o fracaso se basa en la ponderación o peso de los términos. Aunque ha habido mucha investigación sobre técnicas de ponderación de términos, en realidad no hay un consenso sobre cuál método es el mejor. También hay que destacar que el espacio de renglones de la matriz documento-término determinan el contenido semántico de la colección de documentos. Sin embargo, una combinación lineal de dos vectores-documento no representa necesariamente un documento viable de la colección. Más importante aún, mediante el modelo espacio vectorial se pueden explotar las relaciones geométricas entre dos vectores documento (y términos) a fin de expresar las similitudes y diferencias entre términos.

Si bien el rendimiento de un sistema de recuperación de información depende en gran medida de las medidas de similitud entre documentos, la ponderación de términos desempeña un papel fundamental para que esa similitud entre documentos sea más confiable. Así, por ejemplo, mientras que una representación de documentos basada solo en las frecuencias o apariciones de términos no es capaz de representar adecuadamente el contenido semántico de los documentos, la representación de términos ponderados (Aplicación de métodos de normalización a la matriz documento-término) hace frente a errores o incertidumbres asociadas a la representación simple de documentos.

Lectura de Documentos

Primeramente accedemos a la carpeta donde están almacenados los documentos sobre los que realizaremos las consultas, luego antes de agregar cada uno de estos es filtrado por una función *parser* encargada de depurarel archivo devolviendo un vector con los datos necesarios para que sea procesado.

```
def load_data(self):
    os.chdir('/Users/jaime_perez/Documents/School/SRI/SRI/SysRI/corpus/')
    data = [element for element in os.listdir()]
    id = 0
    for d in data:
      id += 1
      file = open(d, 'r', errors='ignore')
      doc = parser.Newsgroup(file)
      self.documents.append(doc_handle.handler(doc[0], doc[1], id))

      file.close()
    os.chdir("..")
    self.index_inverted()
    self.global_weight()
    self.build_dict()
```

Luego como parte de la indexación pasamos a construir la matriz donde para cada termino se le asocian los documentos a los que pertenece llamada matriz de indices invertidos. Posteriormente calculamos y ponderamos el valor de todos los términos en sus respectivos documentos y construimos un diccionario donde tendremos como llave las palabras en su forma nativa de cada termino indexado y como valor las modificaciones indexadas.

```
def global_weight(self):
    for term in self.terms.keys():
      for doc in self.terms[term]:
        weight = self.framework.weight_doc(self.documents, self.terms, doc, term)
        doc.set_weight(term,weight)

  def index_inverted(self):
    for doc in self.documents:
      for term in doc.get_terms().keys():
        if term in self.terms.keys():
          self.terms[term].append(doc)
        else: self.terms[term] = [doc]

  def build_dict(self):
    stemmer = PorterStemmer()
    for term in self.terms.keys():
      stem_term = stemmer.stem(term)
      if stem_term not in self.dict.keys():
        self.dict[stem_term] = [term]
      else:
        if term not in self.dict[stem_term]:
          self.dict[stem_term].append(term)
```

Durante la indexación y ponderación de peso de depuran los términos generalmente irrelevantes como conjunciones, preposiciones etc. Luego para el calculo del peso de un documento es importante saber que debe haber un equilibro entre la cantidad de veces que aparece en un documento y la cantidad de documentos en los que aparece dado que si el termino aparece en muchos documentos no es una palabra distintiva por tanto debe tener menor peso, para esto utilizaremos dos formulas una para calcular la frecuencia en el texto *tf* y otra *idf* para calcular la frecuencia invertida de las ocurrencias en documentos de un termino, siendo el peso de un termino en un documento $w_{i,j} = tf_{i,j} * idf_j$ donde i es el documento y j el termino

```
def tf(self, doc: doc_handle.handler, term):
    if term in doc.get_terms():
      freq = doc.get_terms()[term]
      freq_max = doc.get_terms().most_common(1)[0][1]
      return freq/freq_max

def idf(self, n , N):
    return m.log(N/n)

def weight_doc(self, docs, terms, d, t):
    return self.idf(len(terms[t]), len(docs)) * self.tf(d, t)
```

Consultas

El trabajo con las consultas es análogamente al trabajo con los documentos en pocas palabras tratamos la consulta como un documento y hacemos el mismo análisis y proceso para su indexación. En este caso se introduce la constante de suavizado a permitiendo amortiguar la variación de peso en los términos que ocurren poco

```
def weight_query(self, docs, terms, a, q, t):
    tf = self.tf(q, t)
    idf = self.idf(len(terms[t]), len(docs))
    return (a + (1-a)*(tf)) * idf
```

Luego para para dar una respuesta adecuada debemos calcular la similitud de cada documento con la consulta utilizando la siguiente formula

```
def sim(self, d: doc_handle.handler, q: query.query, dict):
    weight = 0
    sum_d, sum_q = 0, 0
    for t in q.get_weight().keys():
      w_q = q.get_weight()[t]
```

```
      if t in d.get_weight().keys():
        w_d = d.get_weight()[t]
        weight += w_d * w_q
      else:
        stemmer = PorterStemmer()
        for synonymous in dict[stemmer.stem(t)]:
          if synonymous in d.get_weight().keys():
            w_d = d.get_weight()[synonymous]
            weight += w_d * w_q * 0.50
    vec_d, vec_q = list(d.get_weight().values()), list(q.get_weight().values())
    sum_d = np.linalg.norm(vec_d)
    sum_q = np.linalg.norm(vec_q)

    return weight/(sum_d * sum_q)
```

Calculando así el nivel de similitud entre la consulta con cada documento, teniendo en cuenta también los términos que comparten la misma palabra nativa a pesar de que no aparezcan en el documento, pero con menor importancia que si este apareciese

Posteriormente la respuesta a esta consulta serán los muestren un grado mayor a 0.1

```
def query_response(self, query):
    self.query_weight(query)
    ranking = {}

    for doc in self.documents:
      if doc not in ranking: cs = self.framework.sim(doc, query, self.dict)
      if cs > 0.1:
        ranking[doc] = cs
    return sorted(ranking.items(), key=operator.itemgetter(1),reverse=True)

  def rocchio(self, old_q, a , b, c, relevants, not_relevants):
    new_q = self.create_query(old_q.get_text())
    self.query_weight(new_q)
    for t in new_q.get_weight().keys():
      new_q.set_weight(t, a * old_q.get_weight()[t])
```

Retroalimentacion

La retroalimentación en los sistemas de recuperación de información es algo crucial para los usuarios dada por los problemas existentes para hacer una consulta donde se expongan explícitamente las necesidades. En nuestra *API* utilizaremos una Pseudo-retroalimentacion y tomaremos como relevante el documento seleccionado por el usuario para ver y como no relevante el ultimo en la respuesta ofrecida por la aplicación mostrando los 5 documentos mas afines con su selección.

Para esto utilizaremos el algoritmo *Roccio* para el recalculado del peso de los términos y generar la nueva consulta los documentos seleccionados como relevantes y no relevantes teneindo en cuenta la consulta anterior

```
ef rocchio(self, old_q, a , b, c, relevants, not_relevants):
    new_q = self.create_query(old_q.get_text())
    self.query_weight(new_q)
    for t in new_q.get_weight().keys():
      new_q.set_weight(t, a * old_q.get_weight()[t])

    for d in relevants:
      for t in d.get_weight().keys():
        if t in new_q.get_weight().keys():
          new_q.set_weight(t, new_q.get_weight()[t] +
d.get_weight()[t]*(b/len(relevants)))
        else:
           new_q.set_weight(t, d.get_weight()[t])

      for d in not_relevants:
        for t in d.get_weight().keys():
          if t in new_q.get_weight().keys():
            new_q.set_weight(t, new_q.get_weight()[t] -
d.get_weight()[t]*(c/len(not_relevants)))
          else:
           new_q.set_weight(t, d.get_weight()[t])

    return new_q
```