ЛР8: Ранжирование TF-IDF

Задание

Необходимо сделать ранжированный поиск на основании схемы ранжирования TF-IDF. Теперь, если запрос содержит в себе только термины через пробелы, то его надо трактовать как нечёткий запрос, то есть допускать неполное соответствие документа терминам запроса.

Примеры запросов:

- [роза цветок]
- [московский авиационный институт]

Если запрос содержит в себе операторы булева поиска, то запрос надо трактовать как булев, то есть соответствие должно быть строгим, но порядок выдачи должен быть определён ранжированием TF-IDF. Например:

- [роза & цветок]
- [московский & авиационный & институт]

В отчёте нужно привести несколько примеров выполнения запросов, как удачных, так и не удачных.

Метод решения

1. Изучение схемы ранжирования TF-IDF
2. Внесение изменений в индексатор документов
3. Преобразование поиска в ранжированный
4. Проведение анализа полученного решения

Структура файлов

Выходные данные это 4 бинарных файла. Которые представляют прямой и обратный индекс. Обратный индекс – словарь и файл смещения токенов. Прямой индекс содержит информацию о статьях.

Обратный индекс. Файл представляет собой набор из следующих записей <term_hash><offset><count>, где term_hash – это 32 байта, которые мы получаем применением хэш-функции md5 к термину (термины могу быть разного размера, так мы ограничиваем хранимого «слепка» термина — тем самым уменьшая размер файла, однако теряем по времени т.к появляется слой md5) offset количество байт, для отступа от начала - словаря, для того, чтобы начать считывать информацию о документах в которых встречается данный токен, count – это количество записей, которые содержат информацию о термине. Для того, чтобы найти статьи, содержащие определенный термин, необходимо найти его хэш в индексе, считать смещение и count, затем сместиться в файле-словаре на offset байт и считать count записей из словаря. Offset и count занимает 4-8 байта (на моей системе 8байт).

Таким образом, на один термин в обратном индексе приходится 48 байт.

Словарь - вспомогательная структура для обратного индекса. Сначала словарь состоял из 2 файлов — из файла с записями вида - <doc_id><position> и <doc_id><tf><idf> Где doc_id – это 4 байта, который определяют идентификатор документа. Так как количетсво документов влезает в представление unsigned int. Для преобразования Python пердставление int в его байтовое представление, использовалась функцию int.to_bytes. tf, df – это частота слова, и обратная частота документа соответственно, необходим для оценки релевантности результата для запроса.

Однако было принято решение сдлелать 1 файл в формате <doc_id><tf><idf> tf и df занимает по 4 байта каждый. Position – это так же 4 байта, которые определяют позицию вхождения термина в документ. Таким образом на одно вхождение термина в статью приходится 16 байт. Подобное решение оправдано если у нас небольшой индекс, это охранит нам место однако лишит гибкостит т.к мы не можем использовать булев индекс как данные для tf/idf расширения. Таким образом, нам нужно расширить структуру файла.

Прямой индекс содержит информацию о документах в корпусе. Его структура следующая: <doc_id><offset><title_len><url… <title><url>. doc_id – 4 байта, хранящие идентификатор документа. Offset – сдвиг относительно начала словаря, на который нужно переместиться, чтобы считать урл и заголовок, занимает 4 байта. title_len – два байта, содержащие длину заголовка статьи. Url_len – 4 байта, содержащие длину ссылки на статью. title – заголовок статьи с длинной title_len. Url – ссылка на статью, длиной url_len.

Описание ранжирования

Мера TF (term frequency) определяет частоту вхождения токена в документ и заключается в том, что каждому термину, встречающемуся в документе, присваивается вес, зависящий от количества его появлений в документе:

$$tf(t,d)=\frac{n_t}{m}, где\ n_t - число\ вхождений\ термина\ t\ в\ документ, m - общее\ количество\ слов.$$

Затем для коррекции веса терма используется документная частота. Обратная документная частота имеет вид:

$$idf(t,D)=\log\frac{D}{df}, где\ D - общее\ количество\ документов.$$

После этого комбинируются частота термина в документе (tf) и обратная документная частота (idf) для получения веса каждого термина в каждом документе по формуле:

$$tf\,idf(t,d,D) = tf(t,d) \times idf(t,D)$$

Таким образом, релевантность документа d равна сумме вхождений всех терминов запроса в этот документ:

$$Score(q,d) = \sum(tf\,idf(t,d,D))$$

Для реализации подсчёта данной меры был слегка изменён индексатор, в который и был добавлен расчёт tf-idf. Для записи полученных результатов добавлена генерация дополнительного файла invert_index_tf_idf (к файлам, генерация которых реализована при выполнении лабораторной работы по булеву индексу). В нём хранятся числа, значения которых мере tf-idf для соответствующего файла:

Код

```
@logging('Build index with tf-idf extension...')
def frequencyRelevance(pair, with_logger = False):
    def logger():
        print("completed by {:.2f}%...".format(idx / len(index) * 100.))
    complete = 0
    idx = 0
    if with_logger:
        timer = set_interval(logger, 10)
    (index, coef1) = pair
    for token in index:
        idx += 1
        index[token] = (compute_idf(coef1, token, index), compute_tf(index[token]))
    return (index, coef1)
```

```python
def __build_vector_from_query(self, terms):
    count_dict = Counter(terms)
    tf = list(map(lambda term: tf_func(count_dict[term]), terms))
    idfs = [self.index[term][0] if self.index[term] else 1. for term in terms]
    return np.array(tf) * np.array(idfs)

def __build_vectors_from_result(self, terms, result):
    def tf(i):
        res = list(map(lambda v: 1., terms))
        for idx,term in enumerate(terms):
            if self.index[term]:
                for pair in self.index[term][1]:
                    if pair[0] == i:
                        res[idx] = pair[1]
        return res
    idfs = [self.index[term][0] if self.index[term] else 1. for term in terms]
    return [(i, np.array(tf(i)) * np.array(idfs)) for i in result]

def __ranging(self, query, result_list):
    def cosine_similarity(v1, v2):
        return dot(v1, v2)
    terms = list(self.tree.parsed_terms(query, self.__tkn_preprocessing))

    q_vect = self.__build_vector_from_query(terms)
    res_vectors = self.__build_vectors_from_result(terms, result_list)

    return sorted([(i, cosine_similarity(q_vect, v))
        for i, v in res_vectors], key=lambda kv: kv[1], reverse=True)
```

Оценка качества

Так как в задании ничего не сказано про выполнение оценки качества поиска, то дополнительно была выполнена оценка качества, но только по 10 запросам. (Оценки получены с помощью опроса)

№	Текст запроса	Оценка SERP		
			Elastic	Булев
1	Very simple graph algorithm with code example		1,3,1,1,5	3,1,1,1,4
2	Emulate ray movement		1,5,5,1,1	1,1,5,5,3
3	Modern optimization methods		5,4,5,5,5	1,0,0,0,0
4	Old dead programming languages		1,4,2,1,4	0,0,0,0,0
5	Django framework		5,5,4,3,3	1,5,5,0,0
6	(redux | flux | MVVC) & architecture		1,1,1,1,5	3,3,2,1,0
7	Most popular framwork		3,1,5,1,1	1,1,1,2,1
8	Ray tracing		5,1,2,2,1	4,4,4,4,1

9	Popula search engine		4,5,5,5,5	0,0,0,0,0
10	Orithm for parsing polish notation		2,5,1,1,1	1,3,3,5,2

Запрос	**Оценки ранжированного поиска**													
	P			**CG**			**DCG**			**NDCG**			**ERR**		
	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5
1	1.0	0.7	0.4	4.0	8.0	10	4.0	6.39	7.21	0.8	0.4	0.3	0.94	0.95	0.95
2	1.0	0.3	0.2	5.0	8.0	10	5.0	6.76	7.58	1.0	0.5	0.3	0.96	0.97	0.97
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	1.0	0.7	0.4	5.0	10	10	5.0	8.15	8.15	1.0	0.5	0.3	0.97	0.98	0.98
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	1.0	0.7	0.4	5.0	9.0	9.0	5.0	7.4	7.4	1.0	0.5	0.3	0.96	0.97	0.97
8	1.0	1.0	0.8	5.0	11	15	5.0	8.4	10.0	1.0	0.6	0.4	0.96	0.97	0.97
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	1.0	0.7	0.4	4.0	9.0	9.0	4.0	6.89	6.89	0.8	0.5	0.3	0.94	0.95	0.95

Средние значения по 10 запросам

Метрика	Яндекс			Википедия			Булев			Ранжированный		
	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5
P	1.0	0.77	0.76	0.9	0.64	0.62	0.3	0.33	0.28	0.6	0.41	0.26
CG	4.8	11.4	18.2	3.5	9.1	15.3	1.5	5.4	7.2	2.8	5.5	5.4
DCG	5.75	5.4	7.11	2.56	4.41	5.56	1.5	3.67	4.7	2.8	4.56	5.9
NDCG	1.46	0.9	0.73	0.82	0.45	0.41	0.3	0.25	0.19	0.56	0.3	0.19
ERR	0.86	0.97	0.99	0.77	0.87	0.90	0.5	0.52	0.47	0.57	0.58	0.58

Как видно, ранжированный поиск в целом работает лучше булева, однако всё ещё хуже поиска ElasticSearch.

Выводы

В процессе выполнения данной лабораторной работы были внесены изменения в алгоритм индексации корпуса документов и алгоритм поиска, преобразовав его в ранжированный. Полученное решение было проанализировано.