Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Информационный поиск»

Студент: И. А. Мариничев
Преподаватель: А. А. Кухтичев
Группа: М8О-408Б-19

Дата:
Оценка:
Подпись:

Москва, 2023

Лабораторная работа №8 «Ранжирование TF-IDF»

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, т.е. допускать неполное соответствие документа терминам запроса и т.п. Примеры запросов:

- [роза цветок]
- [московский авиационный институт]

Если запрос содержит в себе операторы булева поиска, то запрос надо трактовать как булев, т.е. 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.

1 Описание

Нечёткий поиск я реализовал как объединение всех документов, в которых находятся слова из запроса. Т. е. он происходил аналогично булевому, где вместо дефолтного оператора И используется оператор ИЛИ. Сначала документу ищутся в соответствии с запросом, потом они передаются в функцию, где происходит ранжирование TF-IDF по формуле:

$$rank_d = \sum_{t=1}^{N_{terms}} w_{t,d}$$

$$w_{t,d} = (1 + log10(tf_{t,d})) * log10(\frac{N}{df_t})$$

Ниже приведены несколько запросов и результаты по ним.

Запрос	Время (в ms)	Количество найденных файлов
rose flower	8003.76	4120
moscow aviation institute	13774.9	6032
lord of the rings	174164	46485
who framed roger rabbit	67074.2	28285
abraham lincoln	1365.61	1867

2 Исходный код

Ниже приведена полная реализация класса информационного поиска, обрабатывающего пользовательские запросы и записывающего результаты в файл:

```
// query.h
#include "index.h"

#include <stack>
#include <set>
#include <algorithm>
#include <cmath>

#define AND L'&'
#define OR L'|'
#define NOT L'!'
#define LEFT_BRACKET L'('
#define RIGHT_BRACKET L')'
#define QUOTE L'\"'
#define SLASH L'/'
#define SPACE L' '

struct TFIDF
{
    std::unordered_map<uint32_t, double> ranks;

    TFIDF(std::unordered_map<uint32_t, double> ranks) { this->ranks = ranks; }
    bool operator()(const uint32_t &a, const uint32_t &b) { return ranks[a] > ranks[b]; }
};

class Query
{
public:
    void GetIndex(std::string &inputFile);
    void ParseQueries(std::string &outputFile);
    void ParseQueriesFromFile(std::string &inputFile, std::string &outputFile);

private:
    Index index;
    std::stack<std::shared_ptr<std::vector<uint32_t>>> operands;
    std::stack<wchar_t> operations;

    const std::vector<uint32_t> &GetDocIndices(std::wstring &word);

    void ProcessingQuery(std::wstring &query);
    void ProcessingFuzzyQuery(std::wstring &query);
    void ProcessingQuote(std::wstring &quote);

```

```
    bool IsFuzzy(std::wstring &query);
    void Ranking(std::vector<uint32_t> &result, std::wstring &query);

    bool IsOperation(wchar_t op);
    uint32_t Priority(wchar_t op);

    void ExecuteOperation(wchar_t op);
    void Union();
    void Intersection();
    std::vector<uint32_t> IntersectionForQuote(std::wstring word1, std::wstring word2,
                                               std::vector<uint32_t> &l, std::vector<
                                                   uint32_t> &r,
                                               size_t k);
    void Negation();
};
```

```
// query.cpp
#include "query.h"

void Query::GetIndex(std::string &inputFile)
{
    index.Load(inputFile);
}

void Query::ParseQueries(std::string &outputFile)
{
    std::wstring query;
    bool isFuzzy = false;
    while (std::getline(std::wcin, query))
    {
        if (!query.length())
        {
            break;
        }

        std::wofstream wFileOut(outputFile.c_str());

        isFuzzy = IsFuzzy(query);
        auto start = std::chrono::high_resolution_clock::now();
        if (isFuzzy)
        {
            ProcessingFuzzyQuery(query);
        }
        else
        {
            ProcessingQuery(query);
        }
        std::shared_ptr<std::vector<uint32_t>> result_ptr = operands.top();
        operands.pop();
        if (isFuzzy)
```

```
35          {
36              Ranking(*result_ptr, query);
37          }
38          auto end = std::chrono::high_resolution_clock::now();
39
40          std::cout << "Found " << (*result_ptr).size() << " result(s) in "
41                    << std::chrono::duration<double, std::milli>(end - start).count() << "
                        ms\n";
42
43          wFileOut << (*result_ptr).size() << L'\n';
44          for (const auto &i : (*result_ptr))
45          {
46              wFileOut << index.docIndex[i].title << L' ' << index.docIndex[i].url << L'\
                    n';
47          }
48          wFileOut.close();
49      }
50  }
51
52  void Query::ParseQueriesFromFile(std::string &inputFile, std::string &outputFile)
53  {
54      std::wifstream wFileIn(inputFile.c_str());
55      std::wofstream wFileOut(outputFile.c_str());
56
57      std::wstring query;
58      bool isFuzzy = false;
59      while (std::getline(wFileIn, query))
60      {
61          if (!query.length())
62          {
63              break;
64          }
65
66          isFuzzy = IsFuzzy(query);
67          auto start = std::chrono::high_resolution_clock::now();
68          if (isFuzzy)
69          {
70              ProcessingFuzzyQuery(query);
71          }
72          else
73          {
74              ProcessingQuery(query);
75          }
76          std::shared_ptr<std::vector<uint32_t>> result_ptr = operands.top();
77          operands.pop();
78          if (isFuzzy)
79          {
80              Ranking(*result_ptr, query);
81          }
```

```
82          auto end = std::chrono::high_resolution_clock::now();
83
84          std::cout << "Found " << (*result_ptr).size() << " result(s) in "
85                    << std::chrono::duration<double, std::milli>(end - start).count() << "
                          ms\n";
86
87          wFileOut << (*result_ptr).size() << L'\n';
88          for (const auto &i : (*result_ptr))
89          {
90              wFileOut << index.docIndex[i].title << L' ' << index.docIndex[i].url << L'\
                    n';
91          }
92      }
93      wFileIn.close();
94      wFileOut.close();
95  }
96
97  const std::vector<uint32_t> &Query::GetDocIndices(std::wstring &word)
98  {
99      static const std::vector<uint32_t> empty(0);
100
101     auto it = index.invertedIndex.find(word);
102     if (it != index.invertedIndex.end())
103     {
104         return it->second;
105     }
106     else
107     {
108         return empty;
109     }
110 }
111
112 void Query::ProcessingQuery(std::wstring &query)
113 {
114     std::wstring word;
115     std::wstring quote;
116     for (size_t i = 0; i < query.length(); ++i)
117     {
118         wchar_t c = query[i];
119
120         if (query.substr(i, 4) == L" && ")
121         {
122             c = AND;
123             i += 3;
124         }
125         else if (query.substr(i, 4) == L" || ")
126         {
127             c = OR;
128             i += 3;
```

```
129         }
130         else if (c == SPACE)
131         {
132             c = AND;
133         }
134 
135         if (c == QUOTE)
136         {
137             do
138             {
139                 i++;
140                 c = query[i];
141                 quote += towlower(c);
142             } while (query[i + 1] != QUOTE);
143             i++;
144             ProcessingQuote(quote);
145             quote.clear();
146             continue;
147         }
148 
149         if ((IsOperation(c) || c == LEFT_BRACKET || c == RIGHT_BRACKET) && word.length
                ())
150         {
151             auto postings = std::make_shared<std::vector<uint32_t>>(GetDocIndices(word)
                    );
152             operands.push(postings);
153             word.clear();
154         }
155 
156         if (c == LEFT_BRACKET)
157         {
158             operations.push(LEFT_BRACKET);
159         }
160         else if (c == RIGHT_BRACKET)
161         {
162             while (operations.top() != LEFT_BRACKET)
163             {
164                 ExecuteOperation(operations.top());
165                 operations.pop();
166             }
167             operations.pop();
168         }
169         else if (IsOperation(c))
170         {
171             while (!operations.empty() && (((c != NOT) && (Priority(operations.top())
                    >= Priority(c))) ||
172                                            ((c == NOT) && (Priority(operations.top()) >
                                                Priority(c)))))
173             {
```

```
174                 ExecuteOperation(operations.top());
175                 operations.pop();
176             }
177             operations.push(c);
178         }
179         else
180         {
181             word += towlower(c);
182         }
183     }
184 
185     if (word.length())
186     {
187         auto postings = std::make_shared<std::vector<uint32_t>>(GetDocIndices(word));
188         operands.push(postings);
189     }
190 
191     while (!operations.empty())
192     {
193         wchar_t op = operations.top();
194         ExecuteOperation(op);
195         operations.pop();
196     }
197 }
198 
199 void Query::ProcessingFuzzyQuery(std::wstring &query)
200 {
201     std::wstring word;
202     for (size_t i = 0; i < query.length(); ++i)
203     {
204         if ((query[i] == SPACE) && word.length())
205         {
206             auto postings = std::make_shared<std::vector<uint32_t>>(GetDocIndices(word)
                    );
207             operands.push(postings);
208             word.clear();
209         }
210 
211         if ((query[i] == SPACE))
212         {
213             while (!operations.empty())
214             {
215                 Union();
216                 operations.pop();
217             }
218             operations.push(OR);
219         }
220         else
221         {
```

```
222             word += towlower(query[i]);
223         }
224     }
225 
226     if (word.length())
227     {
228         auto postings = std::make_shared<std::vector<uint32_t>>(GetDocIndices(word));
229         operands.push(postings);
230     }
231 
232     while (!operations.empty())
233     {
234         Union();
235         operations.pop();
236     }
237 }
238 
239 void Query::ProcessingQuote(std::wstring &quote)
240 {
241     std::wstring word = L"";
242     std::wstring wordPrevious;
243     std::stack<std::vector<uint32_t>> result;
244 
245     size_t i = 0;
246     size_t k = 1;
247     while (i <= quote.size())
248     {
249         if ((i == quote.size()) or (quote[i] == SPACE))
250         {
251             if ((word.size() != 0) and (result.size() == 0))
252             {
253                 wordPrevious = word;
254                 result.push(GetDocIndices(word));
255                 word = L"";
256             }
257             if (word.size())
258             {
259                 std::vector<uint32_t> postings1 = result.top();
260                 result.pop();
261                 std::vector<uint32_t> postings2 = GetDocIndices(word);
262 
263                 result.push(IntersectionForQuote(wordPrevious, word, postings1,
                        postings2, k));
264                 k = 1;
265                 wordPrevious = word;
266                 word = L"";
267             }
268         }
269         else if (quote[i] == SLASH)
```

```
270         {
271             k = 0;
272             i++;
273             while ('0' <= quote[i] and quote[i] <= '9')
274             {
275                 k *= 10;
276                 k += (size_t)(quote[i] - '0');
277                 i++;
278             }
279             i--;
280         }
281         else
282         {
283             word += quote[i];
284         }
285         i++;
286     }
287 
288     auto result_ptr = std::make_shared<std::vector<uint32_t>>(
289         result.size() == 1 ? result.top() : std::vector<uint32_t>());
290     operands.push(result_ptr);
291 }
292 
293 bool Query::IsFuzzy(std::wstring &query)
294 {
295     for (size_t i = 0; i < query.size(); i++)
296     {
297         if (query[i] == AND or query[i] == OR or query[i] == NOT or
298             query[i] == LEFT_BRACKET or query[i] == RIGHT_BRACKET or
299             query[i] == QUOTE)
300         {
301             return false;
302         }
303     }
304     return true;
305 }
306 
307 void Query::Ranking(std::vector<uint32_t> &result, std::wstring &query)
308 {
309     std::unordered_map<uint32_t, double> ranks;
310     for (const auto &i : result)
311     {
312         ranks[i] = 0;
313         std::wstringstream wQueryIn(query);
314         std::wstring word;
315         while (getline(wQueryIn, word, L' '))
316         {
317             // double N_d = static_cast<double>(index.docIndex[i].wordCount);
318             double tf_d = static_cast<double>(index.coordinateIndex[word + L' ' + std::
```

```
                to_wstring(i)].size());
319             double N = static_cast<double>(index.docTotal);
320             double df_t = static_cast<double>(index.invertedIndex[word].size());
321             // ranks[i] += (tf_d / N_d) * std::log10(N / df_t);
322             if (tf_d > 0)
323             {
324                 ranks[i] += (1 + std::log10(tf_d)) * std::log10(N / df_t);
325             }
326         }
327     }
328     std::sort(result.begin(), result.end(), TFIDF(ranks));
329 }
330
331 bool Query::IsOperation(wchar_t op)
332 {
333     return (op == AND) || (op == OR) || (op == NOT);
334 }
335
336 uint32_t Query::Priority(wchar_t op)
337 {
338     switch (op)
339     {
340     case OR:
341         return 1;
342     case AND:
343         return 2;
344     case NOT:
345         return 3;
346     default:
347         return 0;
348     }
349 }
350
351 void Query::ExecuteOperation(wchar_t op)
352 {
353     switch (op)
354     {
355     case OR:
356         Union();
357         break;
358     case AND:
359         Intersection();
360         break;
361     case NOT:
362         Negation();
363         break;
364     default:
365         break;
366     }
```

```
367 }
368
369 void Query::Union()
370 {
371     std::vector<uint32_t> result;
372     std::shared_ptr<std::vector<uint32_t>> postings1_ptr = operands.top();
373     operands.pop();
374     std::shared_ptr<std::vector<uint32_t>> postings2_ptr = operands.top();
375     operands.pop();
376     std::set_union((*postings1_ptr).begin(), (*postings1_ptr).end(),
377                    (*postings2_ptr).begin(), (*postings2_ptr).end(),
378                    std::back_inserter(result));
379     auto result_ptr = std::make_shared<std::vector<uint32_t>>(result);
380     operands.push(result_ptr);
381 }
382
383 void Query::Intersection()
384 {
385     std::vector<uint32_t> result;
386     std::shared_ptr<std::vector<uint32_t>> postings1_ptr = operands.top();
387     operands.pop();
388     std::shared_ptr<std::vector<uint32_t>> postings2_ptr = operands.top();
389     operands.pop();
390     std::set_intersection((*postings1_ptr).begin(), (*postings1_ptr).end(),
391                           (*postings2_ptr).begin(), (*postings2_ptr).end(),
392                           std::back_inserter(result));
393     auto result_ptr = std::make_shared<std::vector<uint32_t>>(result);
394     operands.push(result_ptr);
395 }
396
397 std::vector<uint32_t> Query::IntersectionForQuote(std::wstring word1, std::wstring
        word2,
398                                                   std::vector<uint32_t> &l, std::vector<
                                                          uint32_t> &r,
399                                                   size_t k)
400 {
401     std::set<uint32_t> result;
402     size_t i = 0, j = 0;
403     while (i < l.size() && j < r.size())
404     {
405         if (l[i] == r[j])
406         {
407             std::vector<uint32_t> coordinates1 = index.coordinateIndex[word1 + L' ' +
                    std::to_wstring(l[i])];
408             std::vector<uint32_t> coordinates2 = index.coordinateIndex[word2 + L' ' +
                    std::to_wstring(r[j])];
409
410             size_t ii = 0, jj = 0;
411             while (ii < coordinates1.size())
```

```
412             {
413                 while (jj < coordinates2.size())
414                 {
415                     if ((coordinates2[jj] - coordinates1[ii]) > 0 and
416                         (coordinates2[jj] - coordinates1[ii]) <= k)
417                     {
418                         result.insert(l[i]);
419                     }
420                     else if (coordinates2[jj] > coordinates1[ii])
421                     {
422                         break;
423                     }
424                     jj++;
425                 }
426                 ii++;
427             }
428             i++;
429             j++;
430         }
431         else if (l[i] < r[j])
432         {
433             i++;
434         }
435         else
436         {
437             j++;
438         }
439     }
440 
441     return std::vector<uint32_t>(result.begin(), result.end());
442 }
443 
444 void Query::Negation()
445 {
446     std::vector<uint32_t> result;
447     std::shared_ptr<std::vector<uint32_t>> postings_ptr = operands.top();
448     operands.pop();
449     size_t j = 0;
450     for (uint32_t i = 0; i < index.docTotal; ++i)
451     {
452         if (j == (*postings_ptr).size() || i < (*postings_ptr)[j])
453         {
454             result.push_back(i);
455         }
456         else if (i == (*postings_ptr)[j])
457         {
458             ++j;
459         }
460     }
```

```
461     auto result_ptr = std::make_shared<std::vector<uint32_t>>(result);
462     operands.push(result_ptr);
463 }
```

3 Выводы

Выполнив восьмую лабораторную работу по курсу «Информационный поиск», я дополнил поиск ранжированием результатов в порядке убывания метрики TF-IDF. Видно, что время для некоторых запросов со стоп-словами достаточно большое, так как практически каждый документ их включает. В целом нечёткий поиск не сильно замедляет поиск, только на некоторых запросах показывая в разы превосходящие результаты по времени. В принципе это нормально, учитывая, что мы тратим время на сортировку.

Список литературы

[1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Клюшина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))