РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ
Факультет физико-математических и естественных наук
Кафедра информационных технологий

«УТВЕРЖДАЮ»
Заведущий кафедрой
информационных технологий
д.ф.-м.н., проф.
_____Ю.Н. Орлов
«__» _____ 2022 г.

ОТЧЕТ
по лабораторной работе №4

Тема: «Как устроены поисковые системы»
по дисциплине «Компьютерный практикум по ИТ»

Предварительная версия

Выполнил:
Студент группы НПИбд-01-21
Студенческий билет № 1032212280
Генералов Даниил Михайлович

_____
(подпись)

9 апреля 2022 г.

Руководитель:
к.ф.-м.н., доцент кафедры
информационных технологий
Виноградов Андрей Николаевич

_____
(подпись)

«__» _____ 2022 г.

Оценка: _____

МОСКВА
2022 г.

1 Введение

Цель работы: Реализация системы оценки и ранжирования документов по релевантности к запросу.

Задачи работы:

- Изучить алгоритмы определения релевантности документа;
- Составить корпус документов;
- Реализовать алгоритмы оценки релевантности и алгоритм сниппетизации

Содержание

2 Ход работы

2.1 Алгоритмы

Поисковые системы – это программы, которые позволяют находить документы из какого-то корпуса, которые соответствуют запросу. В роли документов могут быть страницы на сайте, или даже в (почти) всей Всемирной сети – для первого существуют модули для движков блогов или как части систем управления базами данных (СУБД), а последним занимаются компании с мировым именем, такие как Google или Yandex.

Эти системы используют много разных метрик и эвристик для того, чтобы определить, какие документы будут наиболее интересны пользователю в зависимости от его запроса, но бывает довольно сложно понять, что это за метрики. Как именно ранжирует документы Google, например, никто не скажет, потому

что это знание дало бы спаммерам инструменты для того, чтобы поставить свои сайты выше по-настоящему интересных в поисковой выдаче. Более того, согласно последним публикациям Google, сейчас они используют технологии вроде глубоких нейросетей, из-за чего природа работы Алгоритма не только неизвестна – она непостижима.

Это подчеркивает, что ранжирование документов по релевантности – это совсем не точная наука, и в процессе выполнения этой лабораторной работы эта идея будет возникать снова и снова.

Для наших задач, поиск документа выполняется, сначала определив какую-то метрику релевантности документа запросу, затем вычислив эту метрику для всех документов, и наконец отсортировав все документы по этой метрике. Для этого мы реализуем две метрики – TF-IDF и метрику близости. После того, как найдены релевантные документы, нужно построить по ним сниппеты – отрезки текста, которые наиболее точно показывают содержимое документа в контексте запроса.

2.1.1 Сниппетизация

Для того, чтобы определить интересный кусок текста, нужно сначала найти все слова, которые есть в запросе и в документе. После этого нужно выбрать какое-то окно, которое содержит эти слова, причем это окно должно быть в каком-то смысле "лучшим".

Поскольку распределение терминов в документе не следует предсказуемой формуле, я решил просто просканировать окном и определить такое, для которого какая-то оценка максимальна. Я решил эту оценку сложить из следующих факторов: IDF ключевого слова; насколько близко ключевое слово находится к центру запроса; сколько раз это слово повторялось в окне. В частности, ключевые слова, находящиеся посередине диапазона ценятся больше всего, с кубическим угасанием к краям окна, и каждое ключевое слово, повторяющиеся больше одного раза, стоит в два раза меньше. Как результат, предпочтение отдается окнам, которые имеют контекст слева и справа от ключевых слов, с некоторым перекосом к началу документа, если ключевые слова повторяются.

```
1 def get_snippet_internal(document, normal_terms, size, skip=10):
2     """
3     Get a snippet from a document corresponding to the given document
  ↪   ID and query.
4     """
5
```

```
    normal_terms = {term.id for term in normal_terms}

    term_idfs = {i: tf_idf_tools.get_term_idf(i) for i in
     ↪  normal_terms}

    document_term_list = [dtp.term for dtp in
     ↪  DocumentTermPosition.select(DocumentTermPosition,
     ↪  NormalizedTerm).join(NormalizedTerm,
     ↪  on=(DocumentTermPosition.term ==
     ↪  NormalizedTerm.id)).where(DocumentTermPosition.document ==
     ↪  document).order_by(DocumentTermPosition.position)]

    best_window_start = -1
    best_window_end = -1
    best_window_score = -1
    best_interesting_indexes = []

    # Iterate over the windows in the document.
    for window_start in range(0, len(document_term_list) - size + 1,
     ↪  skip):
        window_end = window_start + size
        window = document_term_list[window_start:window_end]
        window_score = 0

        # If the term occurs more than once, such a window is
         ↪  exponentially penalized.
        repeat_penalty = collections.defaultdict(lambda: 1)

        interesting_indexes = []

        # Terms are weighted according to their closeness to the
         ↪  center of the window,
        # as well as their IDF.

        for index, term in enumerate(window):
            if term.id in normal_terms:
                interesting_indexes.append(index)
                position = index / len(window)
```

```
                position = 2*position - 1
                term_presence_score = term_idfs[term.id] *
                 ↪  (1-abs(position ** 3)) * repeat_penalty[term.id]
                window_score += term_presence_score
                repeat_penalty[term.id] /= 2

        if window_score > best_window_score:
            best_window_start = window_start
            best_window_end = window_end
            best_window_score = window_score
            best_interesting_indexes = interesting_indexes

    document_word_list =
     ↪  document_stats.split_by_good_words(document.content)
    snippet = document_word_list[best_window_start:best_window_end]
    for i in best_interesting_indexes:
        snippet[i] = '<b>' + snippet[i] + '</b>'
    print(snippet, best_window_score)
    return ' '.join(snippet), best_window_score
```

2.1.2 TF-IDF

TF-IDF – это одна из самых простых функций определения важности слова для документа. Она состоит из произведения двух значений – TF и IDF.

TF (*term frequency*, частота термина) – это мера того, насколько часто встречается данный термин в данном документе, и я решил взять за это число ту пропорцию всех терминов документа, которую составляет этот термин: если этот термин вообще не встречается, то это значение равно 0; если он встречается один раз, то значение равно $\frac{1}{len(doc)}$, а если каждое слово в документе – это соответствующий термин, то значение равно 1.

TF реализован в коде как один SQL `SELECT`-запрос, который выбирает все документы, содержащие один термин, и TF для каждой пары; нужные значения можно затем выбрать из этого запроса.

```
def get_term_tf(term: NormalizedTerm, idf_fac: float = 1) ->
 ↪  pw.ModelSelect:
    """
    Returns a Select query
```

```
    which establishes a mapping between every document
    and TF(term, document).

    `idf_fac` is premultiplied into the result; by default it is 1,
    meaning this returns plain TFs. If the term's IDF is supplied,
    this will return the TF-IDF score instead (though it will still
↪   be called `tf`).

    Column names:

    document_id: the document ID
    tf: the TF for the given term in said document
    document_length: number of terms in the document
    term_count: number of times the given term appears in the
↪   document
    """

    # Select every document;
    # join to it the total number of terms in the document;
    # join to it the number of times our term appears in the
    ↪  document;
    # calculate the ratio between the two (casting the first to float
    ↪  to force float division);
    # finally select the document, the document length, the term
    ↪  count, and the ratio
    tfs = Document.select(
        Document.id.alias('document_id'),
        DocumentTotalTermCount.count.alias('document_length'),
        DocumentTermPairCount.count.alias('term_count'),
        ((
            pw.Cast(DocumentTermPairCount.count, "float") /
            ↪  DocumentTotalTermCount.count)*idf_fac
        ).alias('tf')
    ).join(DocumentTotalTermCount)\
          .switch(Document).join(DocumentTermPairCount)\
            .where(DocumentTermPairCount.term == term)\
            .group_by(Document.id).namedtuples()

```

```
    return tfs
```

IDF (*inverse document frequency*, обратная частота документа) – это мера того, насколько это слово является редким среди всех документов, и определяется как $\log \frac{N}{n_t}$, где N – это количество всех документов, а n_t – количество документов, в которых встретилось это слово. Таким образом, если слово встретилось во всех документах, то мера равна $\log \frac{N}{N} = \log 1 = 0$, а если слово встретилось в единственном документе – то $\log \frac{N}{1} = \log N$ принимает наибольшее значение.

```
@lru_cache(maxsize=1000)
def get_term_idf(term: NormalizedTerm) -> float:
    """
    Returns the IDF for the given term.
    """

    # Get the number of documents that contain this term.
    # This should have been a subquery,
    # but apparently those cannot be inside a CAST,
    # so we compute it here.
    n_containing = DocumentsContainingTermCount.select(
        DocumentsContainingTermCount.count
    ).where(
        DocumentsContainingTermCount.term == term
    ).limit(1).scalar()

    if n_containing is None:
        return 0.0

    # Return the IDF for this term.
    # IDF = log(N / n_containing)

    doc_count = Document.select().count()
    return math.log(
        doc_count / n_containing
    )
```

Произведение этих двух значений для одного документа и одного термина показывает одновременно, насколько это слово важно для этого документа и насколько оно важно по всем документам, и поэтому фактически показывает то,

насколько это слово релевантно этому документу. Для того, чтобы определить релевантность документа и запроса, можно просуммировать TF-IDF для каждого слова в запросе, но более полезной метрикой на практике оказывается близость вектора запроса и вектора документа в общих между ними терминах:

$$S(Q, D) = \frac{\sum_{t \in Q \cap D} tfidf(t, D) \cdot tfidf(t, Q)}{\sqrt{\sum_{t \in Q \cap D} tfidf(t, D)^2} \cdot \sqrt{\sum_{t \in Q \cap D} tfidf(t, Q)^2}}$$

где Q – множество терминов запроса, D – документ, а $tfidf(t, D)$ – метрика релевантности TF-IDF между термином t и набором слов D.

Я реализовал это как один большой `SELECT`-запрос, который выбирает все документы, все слова, и для каждой из пар TF и IDF. Из этого запроса, как раньше, можно выбирать интересные строки и столбцы, чтобы сэкономить вычисление.

```
def get_tf_idf() -> pw.ModelSelect:
    """
    Get the query that contains every pair of documents and terms,
    and their TF-IDF scores.

    To get the TF-IDF score for a term in a document,
    just add a WHERE clause to the query to filter by the document
    ↪ and term.
    If there is no such clause, this will require a full-table scan,
    with a few binary searches based off of that;
    with a WHERE clause, it requires no such scans.

    This also returns a CTE used inside this query, so that it can be
    ↪ bound later.
    """

    # Count the number of documents in the database.
    # This is used to calculate the IDF for each term.
    # I haven't been able to find a way to include this
    # as a subquery in the main query, so I'm just
    # doing it manually.
    count_documents = Document.select().count()

    # Subquery for the TF and IDF metrics separately, as well as the
    ↪  values that go
```

```
    # into both, but separately.
    # As before, this is because I wasn't able to figure out how to combine
    # the two expressions for TF and IDF without repeating them.
    tfs_and_idfs = Document.select(
        Document.id.alias('document_id'),
        NormalizedTerm.id.alias('term_id'),
        DocumentTotalTermCount.count.alias('document_length'),
        DocumentTermPairCount.count.alias('term_count'),
        DocumentsContainingTermCount.count.alias('n_containing'),
        (
            pw.Cast(DocumentTermPairCount.count, "float") / DocumentTotalTermCount.count
        ).alias('tf'),
        (
            pw.fn.Log(
                count_documents / pw.Cast(DocumentsContainingTermCount.count, 'float')
            )
        ).alias('idf'),
        (
            pw.Cast(DocumentTermPairCount.count, "float") / DocumentTotalTermCount.count\
            *
            pw.fn.Log(
                count_documents / pw.Cast(DocumentsContainingTermCount.count, 'float')
            )
        ).alias('tf_idf')
    ).join(NormalizedTerm, join_type=pw.JOIN.CROSS)\
    .switch(Document).join(DocumentTotalTermCount, on=(Document.id == DocumentTotalTermCount.document))\
    .switch(NormalizedTerm).join(DocumentsContainingTermCount, on=(NormalizedTerm.id == DocumentsContainingTermCount.term))\
    .switch(Document).join(DocumentTermPairCount, on=((DocumentTermPairCount.document == Document.id) &
    (DocumentTermPairCount.term==NormalizedTerm.id)))
```

```
    return tfs_and_idfs
```

Я подозреваю, что в поисковых системах вроде Google, помимо члена для важности слова, есть еще член для важности документа – получаемый, возможно, из репутации источника и частоты проходов по ссылке.

2.1.3 Близость

Метрика TF-IDF хорошо помогает определить, что документ содержит важные термины, но не помогает проверить, что они встречаются в составе одной фразы. Для этого можно использовать другую метрику, основаную на близости. Эта метрика не имеет настолько же общепринятого определения, как TF-IDF.

В рамках лабораторной работы я реализовал сначала алгоритм, который перебирает все возможные расположения терминов из запроса и рассматривает их попарное расположение. Более подробно, пусть в запросе есть термины $T_1, T_2, ..., T_n$, а в документе D термин T_n встречается на позициях P_n. Тогда алгоритм выглядит как:

1. Обозначим минимальную сумму расстояний $S \leftarrow \infty$
2. Для каждого $M_1 \in P_1$:
3. Для каждого $M_2 \in P_2$:
4. ...
5. Для каждого $M_n \in P_n$:
6. Обозначим текущую сумму расстояний $\sum \leftarrow 0$.
7. Для каждого $f \in [1; n)$:
8. Для каждого $s \in [f + 1; n]$:
9. $\sum \leftarrow \sum + |f - s|$
10. После завершения шага 7:
11. $S \leftarrow \min(S, \sum)$.

Нетрудно увидеть, что такой алгоритм имеет сложность $O(|D|^n \cdot 2^{|D|})$, что выглядит очень большим и страшным, и действительно на практике вычисление этой метрики для одного документа на лучшем доступном мне компьютере занимает примерно 3 секунды, в зависимости от длины запроса – никак не практично при линейном поиске по документам.

Вопрос на форуме StackOverflow, который я задал, чтобы найти решение этой проблемы (https://stackoverflow.com/q/71724084/5936187), на момент написания этих слов имеет один ответ, отправленный 20 минут назад, который предлагает использование вероятностной модели для определения релевантности (relevance model) – наверняка полезные для дальнейшего улучшения этой программы, но слишком сложные для того, чтобы добавить их в эту лабораторную работу сейчас.

По результатам личной беседы с Виноградовым Андреем Николаевичем мы нашли способ оценить близость терминов, взяв окна, которые начинаются с какого-то ключевого слова, и подсчитывая количество терминов, которые попали в это окно, и их расстояния – тем самым мы находим термин и следующие за ним термины, которые находятся близко друг к другу. Этот алгоритм имеет сложность $O(n)$, что гораздо лучше, поэтому я реализовал его тоже, однако я не смог хорошо сбалансировать различные случаи, чтобы результирующий порядок был хорошим. (Как я уже говорил, настройка этих алгоритмов – совсем не точная наука.)

Однако после этого я вспомнил, что у нас уже есть алгоритм, который находит «хорошее» окно – это алгоритм сниппетизации, который у нас был выше. Можно использовать его же, чтобы оценить, насколько документ релевантен, если просто выполнить его для поиска самого хорошего сниппета и взять его оценку – и поскольку он сканирует документ, его сложность также $O(n)$.

2.2 Корпус и индексы

Для того, чтобы сделать поисковую систему, нужно иметь документы, в которых искать. Параллельно с этой лабораторной работой я занимаюсь проектом анализа данных, в котором требуется набор литературных произведений, поэтому я решил использовать мои наработки в том проекте и использовать документы из *AO3*.

AO3 (полное название – *Archive of Our Own*, https://archiveofourown.org/) – это социальная платформа для авторов и читателей любительских сочинений по мотивам популярных художественных произведений; сокращенно, такие объекты творчества называются "фанфики от английского *fan fiction*. На момент написания этих слов, на этой платформе опубликовано 9,088,424 произведения, а самое популярное по просмотрам произведение – *All the Young Dudes*, автор *MsKingBean89*, https://archiveofourown.org/works/10057010 – имеет 6,416,431 просмотров, 98,188

похвал и 23,690 комментариев. Все эти показатели говорят об активном и обширном сообществе писателей и читателей, что делает его идеально подходящим для задач анализа данных. Вроде этой.

Операторы этой платформы согласны на такое использование своей платформы, при условии соблюдения ограничений по скорости: на запрос про то, как именно стоит скачивать содержимое их сайта, они ответили следующее:

```
Hi, Danya:

Thanks for asking about using the Archive as a data source for your research. We have had similar requests before. If you're planning to use automated means to gather data from the site, our system admins have recommended using a timed curl or https get process to receive the HTML pages without the CSS and JS content. Our system times out requests for works (using the work ID number) that are more frequent than 100 per 400 seconds; search result listings that are more frequent than 100 per 720 seconds; or bookmark result listings more frequent than 30 per 720 seconds. We also ask that you do not scrape on weekends, as those tend to be our busiest times. We do not add specific scraping bots to a permit list, though editing the scraper's user agent to indicate it's a bot may reduce the chance of a summary block.

You should be able to use /works/### - the robots.txt is set to interfere with certain types of filtered search results, hence the '?' in the line.

Be aware that some scrapes will need significant customization to bypass the age verification code and to access works restricted to logged-in users.

Because we are fully staffed by volunteers, we don't have anyone available to compile a database for you.

As well, while our Terms of Service reserve us the right to display the works, the authors of the fanworks maintain all copyrights in their works. As a result, we do not have any ability to give you permission to reproduce any portions of posted works as part of a report. If you want to seek such permissions from the authors of particular fanworks, you can do so by leaving a comment on their works.
```

```
12
13 If you have any questions, let us know!
14
15 Best,
16 CJ
17 AO3 Support
```

Таким образом, я могу скачивать любое количество произведений, если я буду делать это достаточно медленно. К сожалению, учитывая количество произведений, даже если бы для скачивания каждого требовался лишь один запрос (что не так, у многих работ много глав), полное скачивание всех работ потребовало бы 420.7 дня – немного больше, чем дедлайн этой лабораторной работы. Поэтому я взял сравнительно небольшое количество документов – в момент написания этих слов у меня есть 18,018 глав из 4,404 работ, хотя я не знаю точное количество, которое окажется в индексе – и эти документы найдены по верхним результатам поиска по количествам просмотров и похвал, плюс несколько отдельных ключевых слов (определить, что это были за ключевые слова, остается в качестве упражнения читателю).

После того, как присутствуют документы, нужно их проиндексировать. Это выполняет следующая функция.

2.3 Анализ производительности

В статье [**spigot-unbounded**] выдвигается недоказанная гипотеза, что для алгоритма, построенного по ряду Госпера, не требуется проверка того, что выходное значение является верным. Автор статьи пишет, что не смог доказать это утверждение, но проверил его до 1000 знаков. В свою очередь, я проверил его до 1000000 знаков, поэтому считаю, что значения производительности между двумя подходами можно сравнивать напрямую.

Для каждого из графиков, каждое количество знаков π вычислялось заново, не продолжая с предыдущего. В основном это было сделано из-за простоты написания кода, но также из-за того, что тогда счетчик времени не требуется прерывать, тем самым предвнося неточность в измерение.

```
1 @db.atomic()
2 def index_document(document: Document):
3     """
4     Indexes a document, removing all existing relevant indexes.
5
```

```
    This should be called when a document is created or updated.
    """


    # If the document was already in the database, reset its counters.
    # We will recreate them now.
    reset_document_counters(document)

    # Record terms that we have already seen.
    seen_terms = set()

    normalized_terms = []

    # Split the document's content into words.
    words = document.content.split()
    word_natural_term_names = [filter_alnum(word.lower()) for word in words]

    # Select all the natural terms used in the document.
    unique_natural_term_names = set(word_natural_term_names)
    natural_term_name_to_model = dict()
    # Iterate over chunks of the unique set, because there is a limit on the number of parameters in a query.
    for chunk in pw.chunked(unique_natural_term_names, 500):
        natural_term_name_to_model.update(
            {natural_term.name: natural_term for natural_term in NaturalTerm.select().where(NaturalTerm.name.in_(chunk))}
        )


    # For each natural term not in the database, create it.
    natural_terms_to_create = []
    for natural_term_name in unique_natural_term_names:
        if natural_term_name not in natural_term_name_to_model:

            natural_terms_to_create.append(NaturalTerm(name=natural_term
    NaturalTerm.bulk_create(natural_terms_to_create)
```

```
39
40      # Now get the new natural terms.
41      for chunk in
         ↪  pw.chunked(unique_natural_term_names.difference(set(natural_term_nam
         ↪  500):
42          natural_term_name_to_model.update(
43              {natural_term.name: natural_term for natural_term in
                 ↪  NaturalTerm.select().where(NaturalTerm.name.in_(chunk))}
44          )
45
46      # Finally, put the list of natural terms in order of the
         ↪  document.
47      natural_terms = [natural_term_name_to_model[natural_term_name]
         ↪  for natural_term_name in word_natural_term_names]
48
49      # For each natural term, get its corresponding normalized term.
50      # Because there are fewer normalized terms than natural terms,
51      # we rely on the LRU cache around normalize_term.
52      normalized_terms = [normalize_term(natural_term) for natural_term
         ↪  in natural_terms]
53      unique_normalized_terms = set(normalized_terms)
54
55      # For each normalized term, record how many times it's found in
         ↪  the document.
56      dtpc = [DocumentTermPairCount(document=document, term=term,
         ↪  count=normalized_terms.count(term)) for term in seen_terms]
57      DocumentTermPairCount.bulk_create(dtpc)
58
59      # If there were any terms which did not have a corresponding
         ↪  DocumentsContainingTermCount record,
60      # (which means this is a new term first found in this document),
61      # create one for them.
62
63
64      new_terms = []
65      existing_terms = DocumentsContainingTermCount.select().where(
66
             ↪  DocumentsContainingTermCount.term.in_(unique_normalized_terms)
```

```
    )
    existing_terms = set([i.term.id for i in existing_terms])
    for term in seen_terms:
        if term not in existing_terms:
            new_terms.append(term)

    new_dctc_records = [
        DocumentsContainingTermCount(term=natural_term, count=0,
        ↪  last_updated=now())
        for natural_term in new_terms
    ]

    DocumentsContainingTermCount.bulk_create(new_dctc_records)


    # For every term we found, increment the number of documents that
    ↪  contain it.
    DocumentsContainingTermCount.update(
        count=DocumentsContainingTermCount.count + 1,
        last_updated=now()
    ).where(
        DocumentsContainingTermCount.term.in_(seen_terms)
    ).execute()

    # Update the total term count for this document.
    total_terms = len(normalized_terms)
    DocumentTotalTermCount.create(
        count=total_terms,
        last_updated=now(),
        document=document
    )

    # Create the forward index
    term_positions = [
        DocumentTermPosition(
            document=document,
            term=term,
            position=i
```

```
        ) for i, term in enumerate(normalized_terms)
    ]
    DocumentTermPosition.bulk_create(term_positions)
```

(Точный код этой функции мог измениться в интересах рефакторинга и оптимизации.)

Эта функция заносит информацию про документ в базу данных, поэтому сейчас пора показать схему этой базы данных. В качестве ORM-библиотеки используется Peewee, но определения таблиц соотносятся с SQL-таблицами более-менее напрямую.

```
import peewee as pw
import logging

logging.basicConfig(level=logging.DEBUG)

#db = pw.SqliteDatabase('database.db')
db = pw.SqliteDatabase('/run/media/danya/D565-7987/database.db',
 ↪  timeout=15)

class MyModel(pw.Model):
    class Meta:
        database = db


# Special sentinel value for the `last_updated` field
# to force the update of a value.
INVALIDATE = pw.datetime.datetime.min

# Alias for current datetime
now = pw.datetime.datetime.now

def create_table(model: pw.Model):
    """
    Creates a table for a model.
    Return the same model -- that way it can be used as a decorator.
    """
    db.create_tables([model], safe=True)
    return model
```

```python

@create_table
class Document(MyModel):
    """Single instance of document in corpus"""
    title = pw.CharField()
    source = pw.CharField(null=True, index=True)
    content = pw.TextField()
    last_updated = pw.DateTimeField(default=pw.datetime.datetime.now,
    ↪  index=True)

@create_table
class NormalizedTerm(MyModel):
    """Term after normalization and stemming"""
    name = pw.CharField(index=True)

@create_table
class NaturalTerm(MyModel):
    """
    Term as it appears in the document.

    This is what the term looked like before stemming and
    ↪  normalization, but lowercased.
    If a word is not here, it is surely not in any document.
    """
    name = pw.CharField(index=True)
    normalized_form = pw.ForeignKeyField(NormalizedTerm, null=True)

@create_table
class DocumentTermPairCount(MyModel):
    """
    Counts the number of times that this term appears in this
    ↪  document.

    This caches `Document.content.split().count(term)` for each term
    ↪  and `Document`.

    This is used both to calculate the TF-IDF score of a term in a
    ↪  document,
```

```
    and it also forms the inverse index to find documents that contain a term.
    """
    document = pw.ForeignKeyField(Document, on_delete='CASCADE')
    term = pw.ForeignKeyField(NormalizedTerm)
    count = pw.IntegerField()
    last_updated = pw.DateTimeField(default=pw.datetime.datetime.now, index=True)

    class Meta:
        primary_key = pw.CompositeKey('document', 'term')

@create_table
class DocumentsContainingTermCount(MyModel):
    """
    Counts the number of documents that contain this term at least once.

    This caches `len([doc for doc in Document if term in doc.content.split()])` for each term.
    """
    term = pw.ForeignKeyField(NormalizedTerm)
    count = pw.IntegerField(index=True)
    last_updated = pw.DateTimeField(default=pw.datetime.datetime.now, index=True)

    class Meta:
        primary_key = pw.CompositeKey('term')


@create_table
class DocumentTotalTermCount(MyModel):
    """
    Counts the total number of non-unique terms in the document.

    This caches `len(doc.content.split())` for each document.
    """
    document = pw.ForeignKeyField(Document, on_delete='CASCADE')
```

```
94      count = pw.IntegerField()
95      last_updated = pw.DateTimeField(default=pw.datetime.datetime.now,
        ↪  index=True)
96
97      class Meta:
98          primary_key = pw.CompositeKey('document')
99
100 @create_table
101 class DocumentTermPosition(MyModel):
102     """
103     Position of a term in the document.
104
105     Acts like the document's forward index.
106     """
107     document = pw.ForeignKeyField(Document, on_delete='CASCADE')
108     term = pw.ForeignKeyField(NormalizedTerm)
109     position = pw.IntegerField()
110
111     class Meta:
112         primary_key = pw.CompositeKey('document', 'term', 'position')
```

2.4 Анализ результатов

Поскольку мы сделали поисковую систему, а большинство поисковых систем сейчас работают над сущностями из интернета, то логично, что мы будем взаимодействовать с алгоритмом поиска через веб-интерфейс. Это также дает возможность для удобного отображения информациив и дает возможность видеть промежуточные результаты поиска. Последнее достигается путем последовательной передачи кода страницы – сначала отправляется начало страницы, а затем, по мере нахождения результатов, они добавляются в конец страницы, а JS-код позволяет отсортировать результаты по релевантности.

Вычислить IDF для каждого термина, и TF для каждого документа с этим термином, можно очень быстро, через один SQL-запрос и за $O(\log n)$-время. Это можно увидеть на страницах со списками терминов и документов, содержащих термины – эти страницы загружаются так быстро, что преобладающим фактором является время на отображение HTML-кода и скорость сети.

Однако, чтобы отсортировать документы по релевантности, нужно посчитать эту метрику для каждого документа и для каждого термина в запросе. Из-за

того, что это линейный поиск, отображение всех документов по TF-IDF-поиску занимает довольно большое время, где-то вокруг 1200 секунд или 20 минут. Для того, чтобы использовалось меньше времени, можно использовать доступные индексы, чтобы исключать документы как можно раньше: например, известно, что документ может быть релевантен запросу тогда и только тогда, когда в документе встречается как минимум одно слово из запроса, и поэтому поиск по TF-IDF сначала фильтрует документы, которые содержат эти слова.

Поиск по близости также фильтрует документы, но более сильно: для поиска используются только те документы, которые содержат все термины запроса. Это потому, что если какой-то термин не встречается ни разу, то его близость до любого другого термина документа не определена; может быть он вообще не относится к этому документу, или может быть он подходит этому документу, но был пропущен автором. Поэтому, в моей реализации, мы рассматриваем только те документы, которые содержат все слова из запроса.

Для определения количества документов, которые содержат каждый из списка терминов, требуется какой-то сложный запрос, поэтому я вместо этого определяю количество документов, содержащих каждый из терминов запроса, и беру минимум из них – это число точно больше-или-равно настоящему числу таких документов, но оно ближе к истинному значению, чем число всех документов, и поэтому это помогает для более точного отображения прогресса подсчета.

Это важно, потому что алгоритм для определения близости должен просканировать документ скользящим окном, и поэтому каждый один документ обрабатывается медленнее. Однако, поскольку проверяемых документов меньше, то этот алгоритм все равно может быть быстрее TF-IDF для некоторых запросов.

Говоря о скорости запросов, наконец мы переходим к вопросу, сколько времени требуется для полного перечисления результатов по разным типам запросов. Следует заметить, что количество результатов по TF-IDF всегда больше, чем по близости – это потому, что TF-IDF выдает объединение множеств документов по каждому термину, а алгоритм близости, напротив, использует пересечение.

Первые 5 запросов я сочинил, взяв случайные слова сверху списка терминов, поэтому логично, что они редко встречаются вместе. Следующие два запроса – по парам антонимов, которые часто противопоставляются, и поэтому результатов гораздо больше. Следующие 4 запроса составляют списки персонажей из одной оригинальной работы, и последние два из них идентичны, за исключением слово *and*. В том запросе, в котором нет этого слова, есть лишь 2400 ответов, в то время как наличие *and* поднимает это количество до 16474, и поэтому время на полное перечисление результатов различается на два порядка. Количество результатов по близости остается одинаковым, что показывает, что тот метод не настолько чуствителен к присутствию таких аномально частых терминов.

Таким образом, алгоритм TF-IDF непригоден для поиска, содержащего очень частотные ключевые слова, потому что тогда количество допустимых документов стремится к общему количеству документов в корпусе – это можно попытаться решить, игнорируя слова ниже какого-то IDF, но для этого нужно сделать выбор и постоянно его поддерживать, когда в корпус добавляются новые документы.

Последние запросы – буквальные цитаты из определенных документов, где мы ожидаем увидеть эти документы вверху результатов поиска. В колонке результатов в скобках указано, на каком месте в списке результатов оказался документ-источник. То же самое ограничение про стоп-слова применимо здесь – если в запросе есть очень частый термин, то не только поиск будет медленным, но и нужный документ окажется далеко внизу, если использовать TF-IDF-cos. Для запросов, состоящих из цитат или большинства слов из цитат, поиск по близости работает лучше.

По данным нельзя утверждать какое-либо преимущество по скорости между TF-IDF-cos и TF-IDF-$\sum$. Для некоторых запросов один метод работал быстрее, а для других другой, а количество затраченного времени может зависить и от фоновых факторов, поэтому для полноценного анализа потребовалось бы провести много опытов, что затруднительно ввиду количества времени, требуемого для каждого опыта.

Запрос	TF-IDF-cos		TF-IDF-$\sum$		Близость	
	Рез.	Сек.	Рез.	Сек.	Рез.	Сек.
random ghost possession	4350	127.84	4350	132.43	87	50.67
gravitational entanglement	297	4.37	297	3.47	5	4.43
japanese chomper	171	1.50	171	1.55	0	0.45
думал видел сделал	236	8.05	236	4.65	64	12.33
gruesome monstrosity insurance	430	5.05	430	4.45	0	0.53
hot cold	8123	407.49	8123	371.02	2320	457.25
light dark	11665	672.02	11655	1142.87	6635	1037.73
ron hermione harry ginny lucius narcissa remus tonks	1885	42.38	1885	28.00	27	13.78
davion yurnero luna lina mirana terrorblade	618	6.63	618	10.70	0	0.21
mr banks mary poppins	2400	46.79	2400	36.86	4	6.24
mr banks and mary poppins	16474	1132.14	16474	1165.76	4	8.18
automated voice announcer (из документа 26)	10811 (№1)	673.09	10811 (№1)	593.04	18 (№2)	7.85
the wetness of your own blood was dripping down your legs (из документа 5623)	16756 (№≈ 500)	1255.48	—	—	463 (№1)	214.14
rebuilding lost kingdom hyrule (из документа 378)	6914 (№13)	726.84	—	—	4 (№1)	7.18

3 Заключение

В этой лабораторной работе мы реализовали поисковую систему с двумя алгоритмами ранжирования документов – по TF-IDF и по близости. Обе из этих метрики ценны, но лучше работают для разных типов запросов – TF-IDF полезен для поиска документов по ключевым словам, а близость помогает найти буквальные фразы.

Настоящие системы поиска используют сочетание этих двух алгоритмов, а также другие метрики, и они реализуют их гораздо более эффективным способом. Для того, чтобы посмотреть все результаты в моей системе поиска, нужно ждать довольно много времени – перечисление всех 16802 результатов, содержащих слово *a*, занимает около 20 минут с посошью TF-IDF,

А Google, работая по гораздо большему корпусу, находит результаты по любому запросу за доли секунды. Даже после всех изменений, которые их поисковая система притерпела за свою историю, даже несмотря на все пространство, отданное «блокам знаний» (а также и рекламе) – даже после всей оптимизации, чтобы пользователь получил ровно то что нужно, Google все равно пишут, сколько времени нужно было, чтобы ответить на каждый запрос, в шапке каждого результата на настольной версии. Они все еще этим гордятся, потому что тот уровень информатики и сисадминистрирования, который для этого нужен, – это то, пишут докторские диссертации. (Или, раз это коммерческая тайна, то это то, за что дают очень-очень высокие зарплаты.)

Конечно же, я пока что не на таком уровне, но, как эта лабораторная работа подтверждает, такие системы работают на основании конкретных алгоритмов, которые не магические, и понять их все равно возможно. Пусть даже и не сразу.