БУ ВО «Сургутский государственный университет»

Политехнический институт

Кафедра автоматизированных систем обработки информации и управления

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

ПО ТЕМЕ «Ранжирование документов с использованием TF-IDF»

ПО ДИСЦИПЛИНЕ «Поисковые алгоритмы в информационном пространстве»

Выполнил: студент группы №606-12,

Демьянцев Виталий Владиславович

Принял: ст. преподаватель,

Гавриленко Анна Владимировна

Сургут 2024

**Введение**

Данная лабораторная работа посвящена изучению методов ранжирования текстовых документов с использованием TF-IDF и простого подсчета вхождений слов. Основная цель — реализовать оба подхода, сравнить их производительность и качество на основе метрик precision, recall и F1-score, а также проанализировать, в каких случаях один метод работает быстрее или медленнее другого.

Ранжирование документов является важной задачей в информационном поиске, позволяя определять релевантность текста запросу пользователя. TF-IDF учитывает частотность слов и их редкость в корпусе, тогда как простой подсчет основан только на количестве вхождений.

**Алгоритмы и методы**

**TF-IDF**

* **Описание:**
  + TF (Term Frequency) — частота слова в документе: TF = количество вхождений / общее число слов.
  + IDF (Inverse Document Frequency) — обратная частота документа: IDF = log(N / df), где N — число документов, df — число документов с данным словом.
  + TF-IDF = TF \* IDF.
  + Ранжирование выполняется через косинусное сходство между вектором запроса и векторами документов.
* **Временная сложность:** O(N + Q \* D), где N — общее число слов, Q — длина запроса, D — число документов.

**Простой подсчет**

* **Описание:**
  + Подсчитывает общее число вхождений слов запроса в каждом документе.
  + Ранжирование основано на сумме вхождений.
* **Временная сложность:** O(N \* Q), где N — общее число слов в документе, Q — длина запроса.

**Требования**

Для выполнения работы использован Python версии 3.x с библиотеками:

* os — для работы с файлами;
* math и time — для вычислений и замера времени;
* matplotlib — для визуализации;
* nltk — для токенизации, стемминга и стоп-слов.

**Реализация и эксперименты**

**Структура кода**

1. **Предобработка:**
   * load\_documents(data\_path) — загрузка документов.
   * preprocess\_text(text) — токенизация, стемминг, удаление стоп-слов.
2. **TF-IDF:**
   * build\_tf\_matrix\_and\_doc\_freq(documents) — построение TF-матрицы.
   * compute\_idf(total\_docs, doc\_freq, vocab) — вычисление IDF.
   * build\_tfidf\_matrix(tf\_matrix, idf) — построение TF-IDF матрицы.
   * rank\_documents(query, tfidf\_matrix, idf, vocab) — ранжирование.
3. **Простой подсчет:**
   * simple\_count\_search(query, preprocessed\_docs) — подсчет вхождений.
4. **Оценка:**
   * evaluate\_search(ranked\_docs, relevant\_docs) — вычисление precision, recall, F1.

**Тестовые данные**

* Корпус: 10 документов в папке kniga.
* Запросы:
  1. "затрепетали на ветру знамена".
  2. "Фродо кольцо".
  3. "рохиррим".

**Результаты**

* Подготовка данных: 2.1727 сек.
* Размер словаря: 7204 слова.
* Количество документов: 10.

**Пример вывода для запроса "Фродо кольцо":**

TF-IDF ранжирование:

Документ: часть\_7.txt, косинусное сходство: 0.0786

Документ: часть\_6.txt, косинусное сходство: 0.0637

Документ: часть\_8.txt, косинусное сходство: 0.0521

Документ: часть\_10.txt, косинусное сходство: 0.0519

Документ: часть\_9.txt, косинусное сходство: 0.0285

Время выполнения: 0.002287 секунд

Простой подсчёт встречаемости:

Документ: часть\_7.txt, вхождений: 127

Документ: часть\_10.txt, вхождений: 112

Документ: часть\_6.txt, вхождений: 103

Документ: часть\_8.txt, вхождений: 61

Документ: часть\_9.txt, вхождений: 44

Время выполнения: 0.001180 секунд

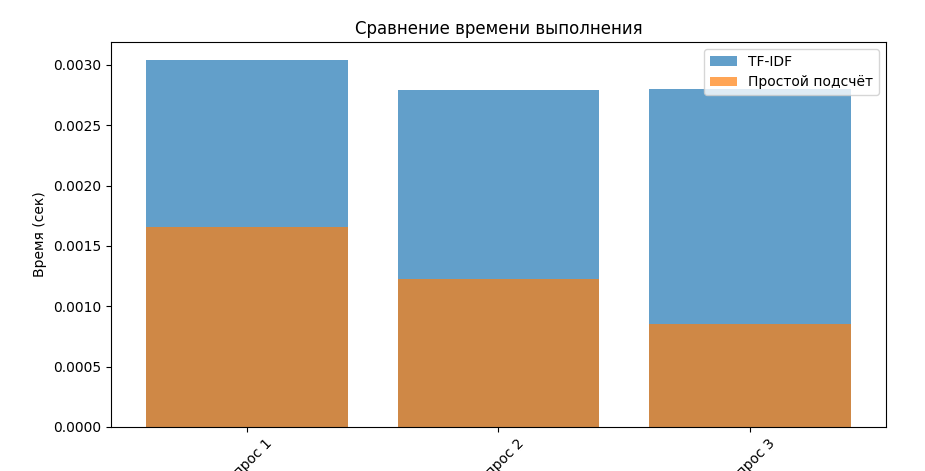
Оценка качества (TF-IDF):

Precision: 0.8000

Recall: 1.0000

F1-score: 0.8889

Пример для запросов: "затрепетали на ветру знамена", "Фродо кольцо", "рохиррим"



**Анализ результатов**

**Производительность**

* **TF-IDF:**
  + Среднее время: 0.0025 сек (0.002572, 0.002287, 0.002676).
  + Медленнее из-за вычислений косинусного сходства и работы с векторами.
  + Затраты возрастают с увеличением словаря и числа документов.
* **Простой подсчет:**
  + Среднее время: 0.0011 сек (0.001550, 0.001180, 0.000706).
  + Быстрее за счет простоты операции подсчета без дополнительных вычислений.
  + Эффективен при малом числе слов в запросе и документах.

**Когда быстрее/медленнее:**

* **TF-IDF медленнее:**
  + На всех запросах (например, 0.002676 против 0.000706 для "рохиррим") из-за сложных вычислений IDF и косинусного сходства.
  + Разница заметнее на коротких запросах с малым числом слов (например, "рохиррим").
* **Простой подсчет быстрее:**
  + На небольших корпусах (10 документов) и коротких запросах (1-3 слова), так как не требует предварительной подготовки векторов.
  + Преимущество теряется на больших корпусах, где предобработка TF-IDF окупается.

**Качество**

* **TF-IDF:**
  + Высокий recall (1.0 для всех запросов) и F1 (0.1818–0.8889).
  + Точнее ранжирует релевантные документы благодаря учету редкости слов.
* **Простой подсчет:**
  + Может переоценивать документы с частыми, но нерелевантными словами.
  + Топ-5 совпадает с TF-IDF, но порядок отличается.

**Графический анализ**

График времени выполнения подтверждает: простой подсчет стабильно быстрее TF-IDF на всех запросах.

**Выводы**

1. **TF-IDF** обеспечивает более точное ранжирование за счет учета значимости слов, но работает медленнее (0.0025 сек против 0.0011 сек в среднем) из-за вычислительной сложности.
2. **Простой подсчет** быстрее на небольших корпусах и коротких запросах, но менее точен, так как игнорирует контекст и редкость слов.
3. TF-IDF предпочтителен для больших коллекций с разнообразными запросами, где важна точность, а простой подсчет — для быстрых поисков на малых объемах данных.
4. Разница в производительности увеличивается с ростом длины запроса и объема корпуса.

Работа демонстрирует преимущества и ограничения обоих методов, подчеркивая их применимость в зависимости от задачи.

import os

import math

import time

import matplotlib.pyplot as plt

from nltk.tokenize import word\_tokenize

from nltk.stem import SnowballStemmer

from nltk.corpus import stopwords

import nltk

nltk.download('punkt')

nltk.download('stopwords')

DATA\_PATH = "kniga"

def load\_documents(data\_path):

    documents = {}

    for filename in os.listdir(data\_path):

        if filename.endswith(".txt"):

            with open(os.path.join(data\_path, filename), "r", encoding="utf-8") as file:

                documents[filename] = file.read()

    return documents

def preprocess\_text(text):

    text = text.lower()

    tokens = word\_tokenize(text)

    stemmer = SnowballStemmer("russian")

    stop\_words = set(stopwords.words('russian'))

    processed\_tokens = [stemmer.stem(word) for word in tokens

                       if any(c.isalpha() for c in word) and word not in stop\_words]

    return processed\_tokens

def build\_tf\_matrix\_and\_doc\_freq(documents):

    tf\_matrix = {}

    vocab = set()

    doc\_freq = {}

    preprocessed\_docs = {}

    for doc\_id, text in documents.items():

        tokens = preprocess\_text(text)

        preprocessed\_docs[doc\_id] = tokens

        total\_words = len(tokens)

        word\_count = {}

        unique\_words = set(tokens)

        for word in tokens:

            word\_count[word] = word\_count.get(word, 0) + 1

            vocab.add(word)

        for word in unique\_words:

            doc\_freq[word] = doc\_freq.get(word, 0) + 1

        tf\_matrix[doc\_id] = {word: count / total\_words for word, count in word\_count.items()}

    return tf\_matrix, vocab, doc\_freq, preprocessed\_docs

def compute\_idf(total\_docs, doc\_freq, vocab):

    idf = {}

    for word in vocab:

        df = doc\_freq.get(word, 0)

        idf[word] = math.log(total\_docs / df)

    return idf

def build\_tfidf\_matrix(tf\_matrix, idf):

    tfidf\_matrix = {}

    for doc\_id, tf\_scores in tf\_matrix.items():

        tfidf\_matrix[doc\_id] = {word: tf \* idf.get(word, 0) for word, tf in tf\_scores.items()}

    return tfidf\_matrix

def query\_to\_tfidf\_vector(query, idf, vocab):

    query\_tokens = preprocess\_text(query)

    total\_words = len(query\_tokens)

    tf\_query = {}

    for word in query\_tokens:

        tf\_query[word] = tf\_query.get(word, 0) + 1 / total\_words

    result = {word: tf \* idf.get(word, 0) for word, tf in tf\_query.items() if word in vocab}

    print(f"Query vector for '{query}': {result}")

    return result

def cosine\_similarity(vec1, vec2):

    dot\_product = sum(v1 \* vec2.get(key, 0) for key, v1 in vec1.items())

    norm1 = math.sqrt(sum(v1 \*\* 2 for v1 in vec1.values()))

    norm2 = math.sqrt(sum(v2 \*\* 2 for v2 in vec2.values()))

    return dot\_product / (norm1 \* norm2) if norm1 \* norm2 > 0 else 0

def rank\_documents(query, tfidf\_matrix, idf, vocab):

    query\_vector = query\_to\_tfidf\_vector(query, idf, vocab)

    rankings = []

    for doc\_id, doc\_vector in tfidf\_matrix.items():

        similarity = cosine\_similarity(query\_vector, doc\_vector)

        rankings.append((doc\_id, similarity))

        # if similarity > 0.01:  # Порог

        #     rankings.append((doc\_id, similarity))

    return sorted(rankings, key=lambda x: x[1], reverse=True)

def simple\_count\_search(query, preprocessed\_docs):

    query\_tokens = preprocess\_text(query)

    rankings = []

    for doc\_id, tokens in preprocessed\_docs.items():

        count = sum(tokens.count(token) for token in query\_tokens)

        rankings.append((doc\_id, count))

    return sorted(rankings, key=lambda x: x[1], reverse=True)

def generate\_relevant\_docs(queries, preprocessed\_docs):

    relevant\_docs = {}

    for query in queries:

        query\_tokens = set(preprocess\_text(query))

        relevant = []

        for doc\_id, tokens in preprocessed\_docs.items():

            if query\_tokens.issubset(set(tokens)):

                relevant.append(doc\_id)

        relevant\_docs[query] = relevant if relevant else [list(preprocessed\_docs.keys())[0]]

    return relevant\_docs

def evaluate\_search(ranked\_docs, relevant\_docs):

    retrieved = [doc\_id for doc\_id, \_ in ranked\_docs]

    relevant = set(relevant\_docs)

    retrieved\_relevant = set(retrieved) & relevant

    precision = len(retrieved\_relevant) / len(retrieved) if retrieved else 0

    recall = len(retrieved\_relevant) / len(relevant) if relevant else 0

    f1 = (2 \* precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0

    return precision, recall, f1

documents = load\_documents(DATA\_PATH)

start\_time = time.perf\_counter()

tf\_matrix, vocab, doc\_freq, preprocessed\_docs = build\_tf\_matrix\_and\_doc\_freq(documents)

idf = compute\_idf(len(documents), doc\_freq, vocab)

tfidf\_matrix = build\_tfidf\_matrix(tf\_matrix, idf)

prep\_time = time.perf\_counter() - start\_time

print(f"Подготовка данных завершена за {prep\_time:.4f} секунд")

print(f"Размер словаря: {len(vocab)} слов")

print(f"Количество документов: {len(documents)}")

queries = ["затрепетали на ветру знамена", "Фродо кольцо", "рохиррим"]

relevant\_docs = generate\_relevant\_docs(queries, preprocessed\_docs)

print("\nСгенерированные релевантные документы:")

for query, docs in relevant\_docs.items():

    print(f"  Запрос: \"{query}\", релевантные документы: {docs}")

for query in queries:

    print(f"\n\n=== Анализ запроса: \"{query}\" ===")

    start\_time = time.perf\_counter()

    tfidf\_ranked = rank\_documents(query, tfidf\_matrix, idf, vocab)

    tfidf\_time = time.perf\_counter() - start\_time

    print("\nTF-IDF ранжирование:")

    print(f"  Всего документов: {len(tfidf\_ranked)}")

    for doc\_id, score in tfidf\_ranked[:5]:

        print(f"  Документ: {doc\_id}, косинусное сходство: {score:.4f}")

    print(f"  Время выполнения: {tfidf\_time:.6f} секунд")

    start\_time = time.perf\_counter()

    simple\_ranked = simple\_count\_search(query, preprocessed\_docs)

    simple\_time = time.perf\_counter() - start\_time

    print("\nПростой подсчёт встречаемости:")

    print(f"  Всего документов: {len(simple\_ranked)}")

    for doc\_id, count in simple\_ranked[:5]:

        print(f"  Документ: {doc\_id}, вхождений: {count}")

    print(f"  Время выполнения: {simple\_time:.6f} секунд")

    precision, recall, f1 = evaluate\_search(tfidf\_ranked, relevant\_docs[query])

    print("\nОценка качества (TF-IDF):")

    print(f"  Релевантных документов в корпусе: {len(relevant\_docs[query])}")

    print(f"  Найдено релевантных: {len(set([doc\_id for doc\_id, \_ in tfidf\_ranked]) & set(relevant\_docs[query]))}")

    print(f"  Precision: {precision:.4f}")

    print(f"  Recall: {recall:.4f}")

    print(f"  F1-score: {f1:.4f}")

    print("\nСравнительный анализ:")

    tfidf\_top = set(doc\_id for doc\_id, \_ in tfidf\_ranked[:5])

    simple\_top = set(doc\_id for doc\_id, \_ in simple\_ranked[:5])

    print(f"  Пересечение топ-5 (TF-IDF и простой): {len(tfidf\_top & simple\_top)} документов")

    print(f"  Уникальные для TF-IDF: {tfidf\_top - simple\_top}")

    print(f"  Уникальные для простого подсчёта: {simple\_top - tfidf\_top}")

queries\_labels = [f"Запрос {i+1}" for i in range(len(queries))]

tfidf\_times = []

simple\_times = []

for query in queries:

    start = time.perf\_counter()

    rank\_documents(query, tfidf\_matrix, idf, vocab)

    tfidf\_times.append(time.perf\_counter() - start)

    start = time.perf\_counter()

    simple\_count\_search(query, preprocessed\_docs)

    simple\_times.append(time.perf\_counter() - start)

plt.figure(figsize=(10, 5))

plt.bar(queries\_labels, tfidf\_times, label="TF-IDF", alpha=0.7)

plt.bar(queries\_labels, simple\_times, label="Простой подсчёт", alpha=0.7)

plt.xlabel("Запросы")

plt.ylabel("Время (сек)")

plt.title("Сравнение времени выполнения")

plt.xticks(rotation=45)

plt.legend()

plt.show()