苏 州 大 学 实 验 报 告

院、系	计算机科学与技术	年级 专业	2020 软件工程	姓名	高歌	学号	2030416018
课程名称	信息检索综合实践					成绩	
指导教师	贡正仙	同组实验者	无		实验日期	2023 年 5 月 3 日	

实 验 名 称　实验 6 文档相似度

一. 实验目的

了解 TF、DF、TF-IDF 等指标的意义及计算方式，学习并实现使用二值向量、TF 向量、TF-IDF 权重矩阵计算文档相似度。

二. 实验内容

针对第 4 次作业中的 10 篇文档，构建向量空间模型，返回 10 篇文档两两相似度。

- 输入：第 4 次作业中的 10 篇文档
- 输出：10 篇文档的两两相似度，并输出与每篇文档最相似的文档号

相似度衡量方法：空间向量法的余弦相似度，即文档用词汇向量表示，相似度使用归一化的余弦相似度表示

计算余弦相似度时，要求汇报以下三种向量表示的余弦相似度

- 用二值法表示词汇向量
- 用对数词频表示词汇向量
- 用 TF-IDF 表示词汇向量

思考

从与某篇最相似的文档角度进行分析

- 上述三种向量表示对相似度计算的影响
- 是不是 TF-IDF 向量表示是这 3 种中最好的？考虑更多的文档或者更少的文档的情况

三. 实验步骤和结果

注：代码使用 TypeScript 编写，运行时使用 ts-node。使用 Prettier 与 ESLint 作为代码格式化工具，代码风格遵从 TypeScript ESLint Recommended 标准。建立索引时使用了 JS 上的 NLP 库 compromise 进行英文分词与词性还原。

（一）实验步骤

1. 本实验逻辑稍有些复杂，这里对目录结构做一个简单解释：

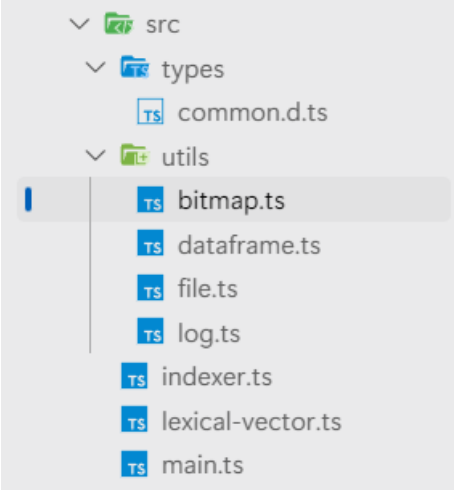

图 1 目录结构

其中，`types`目录下包含了一些必要的类型定义；

`utils`目录下包含了一些工具函数，`bitmap.ts`包含了与位图相关的函数，用来辅助集合操作如去重等，`file.ts`包含了一些与文件相关的工具函数，`log.ts`包含了一些与日志相关的工具函数，而`dataframe.ts`包含了一个用于本次实验的 DataFrame 数据类型定义（类似于 Python 中 Pandas 的 DataFrame）；

而`indexer.ts`包含了构建与读取索引文件所需的相关函数，`lexical-vertor.ts`中包含本次实验中与计算词汇向量有关的几个函数；

`main.ts`则为主入口文件。

其中，`common.d.ts`、`bitmap.ts`、`file.ts`这三个文件的实现与实验 4 完全一致，而`log.ts`的实现与上一次实验（实验 5）完全一致，这里均不再赘述。`indexer.ts`在实验 4 的基础上做了一些细微的改动，会在之后说明。

接下来将一一解释剩余文件中的代码逻辑。

2. 首先解释一下`dataframe.ts`中对 DataFrame 类型的实现。这里先展示其类型定义：

```ts
export interface DataFrame<T> {
  readonly columns: string[];
  readonly rows: string[];

  get(row: string | number, column: string | number): T;
  getRow(row: string | number): T[];
  getColumn(column: string | number): T[];

  set(row: string | number, column: string | number, value: T): void;
  setRow(row: string | number, values: T[]): void;
  setColumn(column: string | number, values: T[]): void;

  copy(): DataFrame<T>;
```

```
  [Symbol.toStringTag]: 'DataFrame';
  toString(options?: {
    limit?: number;
    headerWidth?: number;
    columnWidth?: number;
  }): string;

  saveToFile(pathname: string): Promise<void>;
}

export interface CreateDataFrameOptions {
  columns?: string[];
  rows?: string[];
}

export const createDataFrame = <T>(
  matrix: T[][],
  { columns, rows }: CreateDataFrameOptions = {},
): DataFrame<T> => ...

export const readDataFrame = async <T extends number | string>(
  pathname: string,
): Promise<DataFrame<T>> => ...
```

可以看到，`dataframe.ts`导出了一个用于创建 DataFrame 的函数`createDataFrame`。和 Python 中 Pandas 的 DataFrame 类型相似，该 DataFrame 类型支持使用字符串作为行与列的索引，这有助于之后方便索引单词向量矩阵。

该 DataFrame 类型支持基本的按行、列、单元格的取值与设置值操作，并且支持一个`copy`方法，用于将当前 DataFrame 深复制并创建一个新的 DataFrame。并且其也支持`toString`方法，用于以人类友好的格式打印 DataFrame 中的内容。

并且，该 DataFrame 实现了保存与读取操作，使用`DataFrame#saveToFile`保存，并使用`readDataFrame`函数读取。

该数据类型的具体实现比较繁琐，但并非本实验的重点，因此这里就不展示其完整代码了。

3. 本实验需要分别使用二值向量、对数词频向量与 TF-IDF 向量计算余弦相似度。其中，前两种方式不区分词的重要性，因此考虑使用**去停用词**的索引计算向量矩阵；而对于 TF-IDF 方法，由于其已经考虑了词重要性造成的影响，因此**不去除停用词**。

实验 4 中实现的索引算法构建的索引默认是去除停用词的，因此这里对相关函数稍作修改，使其支持建立不去停用词的索引。修改的部分已经使用不同颜色的背景标出。

```
/**
 * Get lemmatized words from a text.
 */
export const getLemmatizedWords = (
  text: string,
```

```
  {
    includeStopwords = false,
  }: {
    includeStopwords?: boolean;
  } = {},
): string[] => {
  const doc = nlp(normalize(text));

  // Lemmatization
  const result = doc
    .terms()
    .not('#Value')
    .json()
    .map((t: { terms: Array<{ normal: string }> }) => t.terms[0].normal)
    .map(lemmatize)
    .filter((w: string) => /\w/.test(w));

  return includeStopwords ? result : removeStopwords(result);
};

/**
 * Generate the index from the data directory and save it to a file.
 * @param dataDir The data directory.
 * @param pathname The path to the index file.
 * @returns The index map.
 */
export const generateIndex = async (
  dataDir: string,
  pathname: string,
  { includeStopwords = false }: { includeStopwords?: boolean } = {},
): Promise<IndexMap> => {
  const maxDocID = (await fs.readdir(dataDir)).reduce(
    (maxDocID, filename) => Math.max(maxDocID, getDocID(filename)),
    0,
  );

  const wordOccurs = new Map<string, Bitmap>();

  for (const filename of await fs.readdir(dataDir)) {
    const docID = getDocID(filename);
    const content = await fs.readFile(path.join(dataDir, filename),
'utf-8');

    const words = getLemmatizedWords(content, { includeStopwords });
```

```
    ...
```

可以看到，这里添加了一个可选参数`includeStopwords`，表示是否要包含停用词，默认是 false。此外，为了方便后续调用，这里还将 lemmatize 函数单独抽离出来，它接受文档内容，输出文档中所有规范化后的单词（未去重）。

4. 然后在`lexical-vector.ts`中正式开始编写本实验中与词汇向量处理相关的一些函数。首先定义一个辅助函数`indexMap2dataFrame`，用于将索引字典转换成一个空的 DataFrame。

```
/**
 * Convert the document ID map to a data frame.
 * @param indexMap The document ID map, where the key is the term and the
value is the list of document IDs.
 * @returns
 */
const indexMap2dataFrame = (indexMap: IndexMap) => {
  const allDocIDs = getAllDocIDs(indexMap);

  const matrix: number[][] = new Array(indexMap.size)
    .fill(0)
    .map(() => new Array(allDocIDs.length).fill(0));

  return createDataFrame(matrix, {
    columns: allDocIDs.map(String),
    rows: [...indexMap.keys()],
  });
};
```

例如，对于以下的 indexMap：

```
new Map([
  ['john', [1, 3]],
  ['james', [2, 3, 4]],
  ['mary', [2, 5]],
]);
```

将会转换为如图所示的空 DataFrame：

	1	2	3	4	5
john	0.0000	0.0000	0.0000	0.0000	0.0000
james	0.0000	0.0000	0.0000	0.0000	0.0000
mary	0.0000	0.0000	0.0000	0.0000	0.0000

然后，编写生成**二值向量**矩阵的函数：

```
export const generateBinaryLexicalVectorMatrix = (indexMap: IndexMap) =>
{
  const df = indexMap2dataFrame(indexMap);

  for (const [word, docIDs] of indexMap) {
    for (const docID of docIDs) {
```

```
      df.set(word, String(docID), 1);
    }
  }

  return df;
};
```

该函数的原理非常简单，首先通过`indexMap2dataFrame`得到值全为 0 的空 DataFrame，然后遍历索引字典，在出现单词的位置将对应值设为 1 即可。

然后，编写生成**对数词频**矩阵的函数：

```
export const generateLogarithmicWordFrequencyVectorMatrix = (
  indexMap: IndexMap,
  { includeStopwords = false }: { includeStopwords?: boolean } = {},
) => {
  const df = indexMap2dataFrame(indexMap);

  for (const docID of df.columns) {
    const filename = `d${docID}.txt`;
    const content = fs.readFileSync(path.join(DATA_DIR, filename),
'utf-8');
    const words = getLemmatizedWords(content, { includeStopwords });

    for (const word of df.rows) {
      const wordCount = words.filter((w) => w === word).length;
      df.set(
        word,
        docID,
        wordCount === 0 ? 0 : 1 + Math.log(wordCount) / Math.log(10),
      );
    }
  }

  return df;
};
```

该函数的逻辑也不复杂。首先获取一个值全为 0 的空 DataFrame，然后遍历每个文档，读取文档内容，获取文档中的所有单词。然后，对于每个文档，遍历索引字典中的每个单词，计算该单词在该文档中的出现次数，并由此计算出其对数词频保存到 DataFrame 中。

该函数默认调用时不包含停用词，但加入了一个可选参数用于控制是否包含停用词，以供下一个计算 TF-IDF 的函数调用。

然后编写生成 TF-IDF 矩阵的函数：

```
export const generateTFIDFVectorMatrix = (indexMap: IndexMap) => {
  const df = generateLogarithmicWordFrequencyVectorMatrix(indexMap, {
    includeStopwords: true,
```

```
  });

  for (const word of df.rows) {
    const docCount = indexMap.get(word)?.length ?? 0;
    const idf = Math.log(df.columns.length / docCount) / Math.log(10);

    for (const docID of df.columns) {
      const tf = df.get(word, docID);
      df.set(word, docID, tf * idf);
    }
  }

  return df;
};
```

该函数接受一个索引字典（该索引字典应当**包含**停用词）。首先使用`generateLogarithmicWordFrequencyVectorMatrix`函数生成包含停用词的 TF 矩阵，然后遍历矩阵，将每个值乘以 IDF 值即可。

除此之外，还有一个简单的辅助函数用于**计算两个向量的余弦相似度**：

```
export const calculateCosineSimilarity = (
  vector1: number[],
  vector2: number[],
) => {
  const len1 = Math.sqrt(vector1.reduce((sum, v) => sum + v ** 2, 0));
  const len2 = Math.sqrt(vector2.reduce((sum, v) => sum + v ** 2, 0));
  const dotProduct = vector1.reduce((sum, v1, i) => sum + v1 * vector2[i],
0);
  return dotProduct / (len1 * len2);
};
```

该函数首先将两个向量归一化，然后计算它们的点积作为余弦相似度。

11. 最后在`main.ts`中编写主函数。`main.ts`中首先包含了两个辅助函数。第一个辅助函数`calculateConsineSimilarityMatrix`接受一个 DataFrame（向量矩阵），对该矩阵中的每个文档两两比较，得到它们的余弦相似度矩阵。

```
const calculateCosineSimilarityMatrix = (df: DataFrame<number>) => {
  const matrix = new Array(df.columns.length).fill(
    new Array(df.columns.length).fill(0),
  );

  const result = createDataFrame(matrix, {
    columns: df.columns,
    rows: df.columns,
  });
```

```
  for (const docID1 of df.columns) {
    for (const docID2 of df.columns) {
      result.set(
        docID1,
        docID2,
        calculateCosineSimilarity(df.getColumn(docID1),
df.getColumn(docID2)),
      );
    }
  }

  return result as DataFrame<number>;
};
```

然后第二个辅助函数`displayBestMatches`，它接受上面`calculateConsineSimilarityMatrix`函数得到的余弦相似度矩阵，向终端打印对于每个文档相似度最高的文档。

```
const displayBestMatches = (df: DataFrame<number>) => {
  const bestMatches = new Map<string, string>();
  for (const docID of df.rows) {
    const bestMatch = df
      .getColumn(docID)
      .map((value, index) => [value, index])
      .sort(([value1], [value2]) => value2 - value1)[1][1];
    bestMatches.set(docID, df.columns[bestMatch]);
  }

  for (const [docID, bestMatch] of bestMatches) {
    console.log(`Best match for ${docID} is ${bestMatch}`);
  }
};
```

然后编写主函数`main.ts`。

```
export const DATA_DIR = './data';
export const INDEX_PATHNAME = './dict.index';
export const INDEX_PATHNAME_WITH_STOPWORDS =
'./dict.index.with-stopwords';

const BINARY_MATRIX_PATHNAME = './output/binary.csv';
const LOGTF_MATRIX_PATHNAME = './output/logtf.csv';
const TFIDF_MATRIX_PATHNAME = './output/tfidf.csv';
const BINARY_SIMILARITY_MATRIX_PATHNAME =
'./output/binary-similarity.csv';
const LOGTF_SIMILARITY_MATRIX_PATHNAME =
'./output/logtf-similarity.csv';
```

```
const TFIDF_SIMILARITY_MATRIX_PATHNAME =
'./output/tfidf-similarity.csv';

const loggedReadIndex = logged('Index read', readIndex);
const loggedGenerateIndex = logged('Index generated', generateIndex);

const main = async () => {
  const indexMap = (await fileExists(INDEX_PATHNAME))
    ? await loggedReadIndex(INDEX_PATHNAME)
    : await loggedGenerateIndex(DATA_DIR, INDEX_PATHNAME);

  const indexMapWithStopwords = (await fileExists(
    INDEX_PATHNAME_WITH_STOPWORDS,
  ))
    ? await loggedReadIndex(INDEX_PATHNAME_WITH_STOPWORDS)
    : await loggedGenerateIndex(DATA_DIR, INDEX_PATHNAME_WITH_STOPWORDS,
{
        includeStopwords: true,
      });

  /* Generate matrices */
  const binaryMatrix = generateBinaryLexicalVectorMatrix(indexMap);
  if (!(await fileExists(BINARY_MATRIX_PATHNAME)))
    await binaryMatrix.saveToFile(BINARY_MATRIX_PATHNAME);
  console.log('Binary Matrix (without stopwords):');
  console.log(binaryMatrix.toString({ limit: 10 }));
  console.log();

  const logtfMatrix =
generateLogarithmicWordFrequencyVectorMatrix(indexMap);
  if (!(await fileExists(LOGTF_MATRIX_PATHNAME)))
    await logtfMatrix.saveToFile(LOGTF_MATRIX_PATHNAME);
  console.log('LogTF Matrix (without stopwords):');
  console.log(logtfMatrix.toString({ limit: 10 }));
  console.log();

  const tfidfMatrix = generateTFIDFVectorMatrix(indexMapWithStopwords);
  if (!(await fileExists(TFIDF_MATRIX_PATHNAME)))
    await tfidfMatrix.saveToFile(TFIDF_MATRIX_PATHNAME);
  console.log('TF-IDF Matrix (with stopwords):');
  console.log(tfidfMatrix.toString({ limit: 10 }));
  console.log();

  /* Calculate cosine similarity */
```

```
  const binaryCosineSimilarity =
calculateCosineSimilarityMatrix(binaryMatrix);
  if (!(await fileExists(BINARY_SIMILARITY_MATRIX_PATHNAME)))
    await
binaryCosineSimilarity.saveToFile(BINARY_SIMILARITY_MATRIX_PATHNAME);
  console.log('Binary Cosine Similarity Matrix:');
  console.log(binaryCosineSimilarity.toString());
  displayBestMatches(binaryCosineSimilarity);
  console.log();

  const logtfCosineSimilarity =
calculateCosineSimilarityMatrix(logtfMatrix);
  if (!(await fileExists(LOGTF_SIMILARITY_MATRIX_PATHNAME)))
    await
logtfCosineSimilarity.saveToFile(LOGTF_SIMILARITY_MATRIX_PATHNAME);
  console.log('LogTF Cosine Similarity Matrix:');
  console.log(logtfCosineSimilarity.toString());
  displayBestMatches(logtfCosineSimilarity);
  console.log();

  const tfidfCosineSimilarity =
calculateCosineSimilarityMatrix(tfidfMatrix);
  if (!(await fileExists(TFIDF_SIMILARITY_MATRIX_PATHNAME)))
    await
tfidfCosineSimilarity.saveToFile(TFIDF_SIMILARITY_MATRIX_PATHNAME);
  console.log('TF-IDF Cosine Similarity Matrix:');
  console.log(tfidfCosineSimilarity.toString());
  displayBestMatches(tfidfCosineSimilarity);
  console.log();
};

await main();
```

可以看到，主函数首先生成不包含停用词与包含停用词的两个索引，然后按三种算法生成向量矩阵，最后根据三个向量矩阵计算余弦相似度，并展示三种算法中，每个文档对应的相似度最高的文档。

（二）实验结果

运行`npm run dev`，使用 ts-node 执行`./src/main.ts`。下为输出结果：

```
Binary Cosine Similarity Matrix:
                         1       2       3       4       5       6       7       8       9       10
1                   1.0000  0.2384  0.0000  0.0465  0.1131  0.0766  0.1209  0.1188  0.0930  0.0953
2                   0.2384  1.0000  0.0000  0.0488  0.0791  0.0803  0.0845  0.1661  0.0488  0.0500
3                   0.0000  0.0000  1.0000  0.0952  0.2315  0.0000  0.0000  0.0000  0.0000  0.0976
4                   0.0465  0.0488  0.0952  1.0000  0.2315  0.0000  0.0825  0.0810  0.0000  0.0000
5                   0.1131  0.0791  0.2315  0.2315  1.0000  0.0953  0.0668  0.1641  0.0386  0.0395
6                   0.0766  0.0803  0.0000  0.0000  0.0953  1.0000  0.1018  0.1001  0.0392  0.0402
7                   0.1209  0.0845  0.0000  0.0825  0.0668  0.1018  1.0000  0.3158  0.0412  0.0845
8                   0.1188  0.1661  0.0000  0.0810  0.1641  0.1001  0.3158  1.0000  0.0405  0.0415
9                   0.0930  0.0488  0.0000  0.0000  0.0386  0.0392  0.0412  0.0405  1.0000  0.3416
10                  0.0953  0.0500  0.0976  0.0000  0.0395  0.0402  0.0845  0.0415  0.3416  1.0000

Best match for 1 is 2
Best match for 2 is 1
Best match for 3 is 5
Best match for 4 is 5
Best match for 5 is 3
Best match for 6 is 7
Best match for 7 is 8
Best match for 8 is 7
Best match for 9 is 10
Best match for 10 is 9
```

图 2 二值向量计算结果

```
LogTF Cosine Similarity Matrix:
                         1       2       3       4       5       6       7       8       9       10
1                   1.0000  0.2469  0.0000  0.0554  0.1058  0.0799  0.1085  0.1350  0.0848  0.0896
2                   0.2469  1.0000  0.0000  0.0452  0.0749  0.0958  0.0951  0.1856  0.0450  0.0475
3                   0.0000  0.0000  1.0000  0.1209  0.2378  0.0000  0.0000  0.0000  0.0000  0.0915
4                   0.0554  0.0452  0.1209  1.0000  0.2453  0.0000  0.0742  0.0875  0.0000  0.0000
5                   0.1058  0.0749  0.2378  0.2453  1.0000  0.1062  0.0614  0.1638  0.0360  0.0380
6                   0.0799  0.0958  0.0000  0.0000  0.1062  1.0000  0.0998  0.1236  0.0461  0.0487
7                   0.1085  0.0951  0.0000  0.0742  0.0614  0.0998  1.0000  0.3667  0.0369  0.0780
8                   0.1350  0.1856  0.0000  0.0875  0.1638  0.1236  0.3667  1.0000  0.0352  0.0371
9                   0.0848  0.0450  0.0000  0.0000  0.0360  0.0461  0.0369  0.0352  1.0000  0.3341
10                  0.0896  0.0475  0.0915  0.0000  0.0380  0.0487  0.0780  0.0371  0.3341  1.0000

Best match for 1 is 2
Best match for 2 is 1
Best match for 3 is 5
Best match for 4 is 5
Best match for 5 is 4
Best match for 6 is 8
Best match for 7 is 8
Best match for 8 is 7
Best match for 9 is 10
Best match for 10 is 9
```

图 3 对数词频计算结果

```
TF-IDF Cosine Similarity Matrix:
                         1       2       3       4       5       6       7       8       9       10
1                   1.0000  0.1066  0.0008  0.0120  0.0382  0.0145  0.0395  0.0293  0.0169  0.0621
2                   0.1066  1.0000  0.0018  0.0131  0.0136  0.0146  0.0266  0.0438  0.0283  0.0327
3                   0.0008  0.0018  1.0000  0.0436  0.1214  0.0011  0.0055  0.0277  0.0056  0.0540
4                   0.0120  0.0131  0.0436  1.0000  0.1383  0.0027  0.0688  0.0570  0.0068  0.0003
5                   0.0382  0.0136  0.1214  0.1383  1.0000  0.0209  0.0298  0.0791  0.0212  0.0030
6                   0.0145  0.0146  0.0011  0.0027  0.0209  1.0000  0.0505  0.0262  0.0028  0.0028
7                   0.0395  0.0266  0.0055  0.0688  0.0298  0.0505  1.0000  0.2518  0.0060  0.0272
8                   0.0293  0.0438  0.0277  0.0570  0.0791  0.0262  0.2518  1.0000  0.0135  0.0026
9                   0.0169  0.0283  0.0056  0.0068  0.0212  0.0028  0.0060  0.0135  1.0000  0.1547
10                  0.0621  0.0327  0.0540  0.0003  0.0030  0.0028  0.0272  0.0026  0.1547  1.0000

Best match for 1 is 2
Best match for 2 is 1
Best match for 3 is 5
Best match for 4 is 5
Best match for 5 is 4
Best match for 6 is 7
Best match for 7 is 8
Best match for 8 is 7
Best match for 9 is 10
Best match for 10 is 9
```

图 4 TF-IDF 计算结果

四. 实验总结

1. 本次实验中，仿照 Python 中 Pandas 的 DataFrame 建立了一个辅助的 DataFrame 数据结构以辅助矩阵计算。

2. 从本次实验结果中可以看出，TF-IDF 向量表示在三种算法中取得了相对较好的结果。对于包含了大量长文档的情况，TF-IDF 应当能输出更可靠的结果。然而对于本实验中文档较少且内容较短的情况，优势似乎不够明显。并且，TF-IDF 显然具有需要更大计算量的缺点。

事实上，在本题给出的数据中，去除停用词的二值向量与对数词频算法都取得了不错的结果。而这两种算法的时间效率都较高，尤其是二值向量法，由于其不需要遍历文档，因此效率非常高。然而，对于文档较大的情况，二值向量法应当无法取得较好的效果，因为许多文档都可能包含相似的单词。

而对数词频法考虑到了单词在文档中出现的频率。但是，对于常见停用词，这些词语会占用较大的权重，使结果不精确，因此需要去除停用词以获得更精准的结果。相比二值向量法，对数词频法更加精确，但是即使去除了停用词也存在一定的常见词污染问题。比如文档集中的文档全部是对于某个专业领域的研究论文，那么某些专业词汇应当在大多数文档中都具有较大的权重，这会对该算法的精准度造成影响。

TF-IDF 法考虑了单词重要性，不需要去除停用词，并且事实上也具有较高的算法效率（如果存在一个良好的倒排索引）。在这三种算法中，TF-IDF 法应当是最为鲁棒的算法。

3. 通过本次实验，了解并实践了文档相似度的计算方式，为后面的实验做了铺垫。