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 10 日

实验名称 实验 7 基于 TF-IDF 向量表示的信息检索

一. 实验目的

结合上一次实验学习的 TF-IDF 文档相似度，实现在大量文档中检索与某一文档相似度最高的几个文档，得到查询结果。

二. 实验内容

数据文件

下载 en.txt，该文件共 30 万句，要求将每句话当作是一个文档（所以本次实验相当于提供了 30 万个文档）。

备注：en.txt 已作 tokenize，也已作小写处理。

检索准备

（1）为这 30 万个文档构建倒排索引。这一步不要考虑停用词。

（2）在建好的倒排索引基础上，联合每篇文档，计算词汇的 TF-IDF 值。

- 注意：计算每个词汇的 TF-IDF 值时，TF 的值是对每个文档来讲的，而 IDF 的值是对整个数据集（30 万）来讲的。
- 做（1）和（2）时，可以增加词形还原操作，但这里若作了词形还原，后面查询一定也要做，否则会造成漏检。
- 词汇计算好 TF-IDF 值后，可以根据 TF-IDF 值过滤掉一些停用词和符号。

（3）构建文档向量空间。将每篇文档表示成由 TF-IDF 值构成的向量空间。

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

分别输入以下 3 条查询，获得查询结果

- gsk controls us aids Weinstein statement
- china nepal third world
- lopes Barroso

提示 1：要将查询也表示成向量空间，运用下面公式。也可以先将向量归一化，直接用求向量点积的方式（参考作业 6）

$$\cos(\vec{q},\vec{d}) = \textsc{sim}(\vec{q},\vec{d}) = \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

提示 2：如果前面的文档向量表示时进行了词汇过滤和词形还原，那么查询也要做相应操作，否则这里有可能出现文档包含查询词但查询不到的情况（比如查询中包含的所有词都是停用词），请在报告里给出你的解决方案和理由。

输出要求

为以上查询返回相关文档，按照向量相似度进行排序。

（1）每个查询显示最多 5 个相似度排在前面的相关文档，按向量相似的降序排列。

（2）查询结果显示要求：先显示相似度分值，然后显示文档编号（从 1 开始到 30 万）、文档内容。

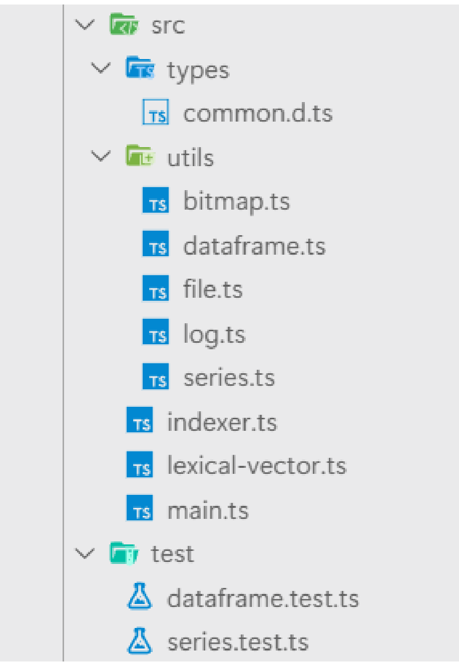
（3）如果无相关文档，返回编号-1，文档内容为 null.

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 进行英文词形还原。

（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：



```
src
  types
    common.d.ts
  utils
    bitmap.ts
    dataframe.ts
    file.ts
    log.ts
    series.ts
  indexer.ts
  lexical-vector.ts
  main.ts
test
  dataframe.test.ts
  series.test.ts
```

图 1 目录结构

其中，`types/`目录下包含了一些必要的类型定义，实际上仅包含了有关索引字典的类型定义；

`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），series.ts 则供 dataframe.ts 使用，作为其内部数据的存储方式；

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，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 6）基本一致，只是改了一些 API 与修了一些 BUG，这里均不再赘述，log.ts 的详细实现将在附录中给出。indexer.ts 在上次实验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. common.d.ts 中定义了本次实验使用的两个索引的类型。实际上本次实验只需要一个索引，这里为了方便起见复用了上次实验的索引实现，但是由于上次实验构建的索引太过复杂，需要进行简化，因此这里创建了一个新类型 SimplifiedIndexMap。

```typescript
/**
 * The sentence ID map, where the key is the term,
 * and the value is a map where the key is the sentence ID
 * and the value is the list of indexes (start by 0) of the term in the sentence.
 *
 * @example
 * ```typescript
 * const indexMap = new Map([
 *   ['john', {
 *     1: [0, 12], // The word 'john' appears twice in the sentence with ID 1, at index 0 and 12.
 *   }],
 * ]);
 * ```
 */
export type IndexMap = Map<string, Record<number, number[]>>;

/**
 * The simplified sentence ID map, where the key is the term,
 * and the value is a map where the key is the sentence ID
 * and the value is the number of times the term appears in the sentence.
 *
 * @example
 * ```typescript
 * const indexMap = new Map([
 *   ['john', {
 *     1: 2, // The word 'john' appears twice in the sentence with ID 1.
 *   }],
 * ]);
```

````
 * ```
 */
export type SimplifiedIndexMap = Map<string, Map<number, number>>;
````

3. 然后介绍一下`series.ts`与`dataframe.ts`。这两个文件实现了一个类似于Python中Pandas库提供的`DataFrame`，其使用`Series`存储数据（按列存储）。这两个文件的代码量相当大，含文档加起来约1800行，并且包含数百行的测试代码，不适合直接在这里给出，将在附录中给出其详细实现。

在这里，仅仅给出它们类型定义中比较重要的几个方法及示例，便于解释之后的代码。

首先是`DataFrame`必要的一些类型定义：

```
/**
 * A DataFrame is a 2-dimensional data structure that can store data of
 * different types (including characters, integers, floating point values,
 * categorical data and more) with labeled columns and (optionally labeled) rows.
 *
 * The data is stored by columns, and each column is a {@link Series},
 * so you should access the data by column if possible to avoid performance issues.
 * @template TData The type of the data stored in the DataFrame.
 * @template TColumnNames The type of the column names, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `col`
 * to `Series<TData, TRowNames, TType> | undefined`.
 * @template TRowNames The type of the row names, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `row`
 * to `Series<TData, TColumnNames, TType> | undefined`.
 * @template TType The type of the DataFrame, can be either `'normal'` or `'sparse'`.
 *
 * @see {@link createDataFrame}
 */
export interface DataFrame<
  TData,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
> {
  /**
   * The type of the DataFrame, can be either `'normal'` or `'sparse'`.
   */
  readonly type: TType;
  /**
   * ...
   */
  readonly shape: [TRowNames['length'], TColumnNames['length']];
```

```
  readonly columnNames: TColumnNames;
  readonly rowNames: TRowNames;

  /**
   * Get or set a column at the given column name (not index).
   *
   * **Typesafety is ensured** when `TColumnNames` is the fallback type `string[]`,
   * i.e. the type of the return value is `Series<TData, TRowNames, TType> | undefined`.
   * While in exact type, the type of the return value is `Series<TData, TRowNames, TType>`.
   *
   * @example
   * ```typescript
   * const df = createDataFrame({
   *   data: [[1, 2, 3], [4, 5, 6]],
   *   columnNames: ['A', 'B', 'C'],
   * });
   * df.col['B']; // Series([2, 5])
   * df.col['B'] = series([42, 43]); // `df` is now [[1, 42, 3], [4, 43, 6]]
   * df.col['B']; // Series([42, 43])
   * ```
   */
  col: {
    [ColumnName in TColumnNames[number]]: string[] extends TColumnNames
      ? Series<TData, TRowNames, TType> | undefined
      : Series<TData, TRowNames, TType>;
  };
  /**
   * Get or set a column at the given integer position of the column,
   * i.e. column index (not name).
   *
   * Typesafety is **not** ensured when `TColumnNames` is the fallback type `string[]`,
   * so the type of the return value is always `Series<TData, TRowNames, TType>`.
   * You have to ensure the index is valid by yourself.
   *
   * @example
   * ```typescript
   * const df = createDataFrame({
   *   data: [[1, 2, 3], [4, 5, 6]],
   *   columnNames: ['A', 'B', 'C'],
   * });
   * df.icol[1]; // Series([2, 5])
   * df.icol[1] = series([42, 43]); // `df` is now [[1, 42, 3], [4, 43, 6]]
   * df.icol[1]; // Series([42, 43])
   * ```
```

```
   */
  icol: {
    [key: number]: Series<TData, TRowNames, TType>;
  };

  /**
   * ...
   */
  show(options?: DataFrameToStringOptions): void;
  /**
   * ...
   */
  showHead(limit?: number): void;

  /**
   * Save the DataFrame to a file.
   * @param pathname Path to the file to save.
   */
  saveToFile(pathname: string): Promise<void>;
}
```

注意，这里只给出了 DataFrame 下面会用到的一些方法定义，并非全部，并且省略了大部分注释与文档。详细实现将在附录中给出。

可以看到，DataFrame 支持通过类似 df.col['c1']的语法访问某一列（得到一个 Series），通过类似 df.col['c2'] = series([10, 20])的语法设置某一列。同时，它支持一些常规的方法，比如保存到文件、以人类友好的方式展示内容、这里未展示的克隆、获取和设置行、矩阵转置、重新设置行名/列名等。

然后展示 Series 下面会用到的一些方法定义：

```
/**
 * A series is a one-dimensional array with labels.
 * @template TData The type of the data.
 * @template TLabels The type of the labels, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `at`
 * to `TData | undefined`.
 * @template TType The type of the series, either `'normal'` or `'sparse'`.
 * A sparse series internally uses a `Record<number, TData>` (an object)
 * to store data, while a normal series uses a `TData[]` (an array).
 *
 * @see {@link createSeries}
 */
export interface Series<
  TData,
  TLabels extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
> {
  /**
   * The type of the series, either `'normal'` or `'sparse'`.
   */
  readonly type: TType;
  /**
   * ...
   */
  readonly length: TLabels['length'];
  /**
   * The labels of the series.
   */
  readonly labels: TLabels;

  /**
   * Get or set the value at the given label (not index).
   *
   * **Typesafety is ensured** when `TLabels` is the fallback type `string[]`,
   * i.e. the type of the return value is `TData | undefined`.
   * While in exact type, the type of the return value is `TData`.
   *
   * @example
   * ```typescript
   * const ser = createSeries({
   *   data: [1, 2, 3, 4, 5],
   *   labels: ['a', 'b', 'c', 'd', 'e'],
   * });
   * ser.at['b']; // 2
   * ser.at['b'] = 42; // `ser` is now [1, 42, 3, 4, 5]
   * ser.at['b']; // 42
   * ```
   */
  at: {
    [Label in TLabels[number]]: string[] extends TLabels
      ? TData | undefined
      : TData;
  };
  /**
   * Get or set the value at the given integer position, i.e. index (not label).
   *
   * Typesafety is **not** ensured when `TLabels` is the fallback type `string[]`,
   * so the type of the return value is always `TData`.
   * You have to ensure the index is valid by yourself.
   *
```

```typescript
   * @example
   * ```typescript
   * const ser = createSeries({
   *   data: [1, 2, 3, 4, 5],
   *   labels: ['a', 'b', 'c', 'd', 'e'],
   * });
   * ser.iat[1]; // 2
   * ser.iat[1] = 42; // `ser` is now [1, 42, 3, 4, 5]
   * ser.iat[1]; // 42
   * ```
   */
  iat: { [key: number]: TData };

  /**
   * Transform the series by applying the given transformer to each value,
   * similar to `Array#map`.
   * @param transformer The callback function to transform each value.
   *
   * @example
   * ```typescript
   * const ser = createSeries({
   *   data: [1, 2, 3, 4, 5],
   *   labels: ['a', 'b', 'c', 'd', 'e'],
   * });
   * const newSer = ser.transform((value) => value * 2); // [2, 4, 6, 8, 10]
   *
   * const sparseSer = createSeries({
   *   type: 'sparse',
   *   length: 3,
   *   labels: ['a', 'b', 'c'],
   * });
   * sparseSer['a'] = 1;
   * sparseSer['c'] = 3;
   * const newSparseSer = sparseSer.transform((value) => value * 2); // [2, <empty>, 6]
   * ```
   */
  transform<NTData>(
    transformer: (
      value: TData,
      index?: number,
      label?: TLabels[number],
      series?: Series<TData, TLabels, TType>,
    ) => NTData,
  ): Series<NTData, TLabels, TType>;
```

```
  /**
   * Accumulate the series by applying the given accumulator to each value,
   * similar to `Array#reduce`, but `initialValue` can be omitted by using
   * the first value of the series as the initial value.
   * @param accumulator The callback function to accumulate each value.
   * @param initialValue The initial value of the accumulator. If omitted,
   * the first value of the series would be used as the initial value.
   * @returns The accumulated value.
   */
  accumulate<NTData>(
    accumulator: (
      acc: NTData,
      value: TData,
      index?: number,
      label?: TLabels[number],
      series?: Series<TData, TLabels, TType>,
    ) => NTData,
    initialValue: NTData,
  ): NTData;
}
```

同样的，这里也只展示了 `Series` 的部分方法，并省略了其大部分文档及注释。详细实现将在附录中给出。

可以看到，与 `DataFrame` 类似，`Series` 通过 `at` 按行名获取和设置值，通过 `iat` 按行号（从 0 开始）获取和设置值。可以使用类似 `ser.at['d']`的语法获取值，使用类似 `ser.at['d'] = 42` 的语法设置值，`iat` 也同理，不多赘述。

考虑到 DataFrame 的`.col['<列名>']`和`.icol[<列号>]`返回一个 Series，因此可以直接通过类似`df.col['c1'].at['r1']`的语法获取位于某行某列的值，通过类似`df.col['c1'].at['r1'] = 10` 的语法设置位于某行某列的值。当然，也可以将它们与`.icol[<列号>]`和`.iat[<行号>]`混搭使用，如`df.icol[0].at['r1']`、`df.icol[0].iat[2]`等。

注：实际上，在最初设计相关 API 时，还存在一个更简洁的语法，支持直接使用类似`df['c1']['r1']`的方式获取值，使用类似`df['c1']['r1'] = 10` 的方式设置值，这与 Pandas 非常类似。但是 JS 中通过方括号访问实际上等价于通过点号访问，即 `df['c1']['r1']`事实上等价于`df.c1.r1`，而这很容易造成命名冲突，考虑若存在某个命名为`'length'`甚至`'toString'`的行名或列名，因此弃用了该语法。

此外，`Series` 也支持 `transform` 和 `accumulate` 方法，它们的命名灵感来自于 C++ 11 中的两个同名函数，可以理解为 JS 中数组上的 `Array#map` 与 `Array#reduce` 两个方法。之所以不直接命名为 `map` 和 `reduce`，是为了避免与数组上的相应方法产生混淆。

例如，`series([1, 2, 3]).transform((val) => val + 1)`得到`series([2, 3, 4])`，而`series([1, 2, 3]).accumulate((acc, val) => acc + val, 0)`得到`6`，

4. 下面介绍 `indexer.ts` 与索引相关的一些改动。

首先将词形还原的函数即 `lemmatize` 抽取了出来，供之后的查询使用。原本依赖于该逻辑的函

数 generateIndex 除了这处外并无其他改动，这里不详述。

```
/**
 * Lemmatize a term (verb -> infinitive, noun -> singular).
 * @param term The term to lemmatize.
 * @returns The lemmatized term.
 */
export const lemmatize = (term: string): string => {
  const t = nlp(term);
  if (t.has('#Verb')) {
    return t.verbs().toInfinitive().out('text');
  }
  if (t.has('#Noun')) {
    return t.nouns().toSingular().out('text');
  }
  return term;
};
```

然后比较奇怪的是，实验所给的文档似乎存在一些不正常的字符，如字符¡，因此建立了一个函数 preprocess 用于预处理一下所给的文档。

```
/**
 * Preprocess the text by removing meaningless characters and extra spaces.
 * @param text The text to preprocess.
 * @returns The preprocessed text.
 */
const preprocess = (text: string): string =>
  text
    .replaceAll('¡¯s', " 's")
    .replace(/¡.?/g, ' ')
    .replace(/\s+/g, ' ')
    .trim();
```

因此目前的 generateIndex 函数大致是这样：

```
export const generateIndex = async (
  documents: string[],
  {
    lemmatize: doesLemmatize = true,
    logging = true,
  }: GenerateIndexOptions = {},
): Promise<IndexMap> => {
  const indexMap = new Map<string, Record<number, number[]>>();

  let lastLoggingInfo = '';
  for (const [index, document] of documents.entries()) {
    const docID = index + 1;

    const processedLine = preprocess(document);
    const words = processedLine
```

```
      .split(' ')
      .filter((w) => /^[a-z]+$/.test(w))
      .map(doesLemmatize ? lemmatize : R.identity);

      ...
```

其他地方与上次实验相比没有明显改动。

然后新增了一个函数 transformIndexMap，用于将索引字典转换为简化的索引字典（为了加快之后计算 TF-IDF 矩阵的效率）。这一步骤原本是不必要的，实际上完全可以修改 generateIndex、saveIndex 与 loadIndex（原名 readIndex，这里只是改了个名字）的实现以使其生成和读取简化后的索引字典，这里只是为了复用上次的代码而添加了一个简化层。

```
/**
 * Transform the index map to a simplified index map.
 * @param indexMap The index map.
 * @returns The simplified index map.
 */
export const transformIndexMap = (indexMap: IndexMap): SimplifiedIndexMap =>
  new Map(
    [...indexMap.entries()].map(([word, value]) => [
      word,
      Object.entries(value).reduce((acc, [sentenceID, positions]) => {
        acc.set(Number.parseInt(sentenceID), positions.length);
        return acc;
      }, new Map<number, number>()),
    ]),
  );
```

然后为 loadIndex（原名 readIndex）新增了一个选项，可以指定是否对返回值调用 transformIndexMap.

```
/**
 * Load the index from a file.
 *
 * If `simplified` is `true`, the index map will be transformed to a simplified
 * index map by {@link transformIndexMap}.
 * @param pathname The path to the index file.
 * @param options The options.
 * @returns The index map.
 */
export async function loadIndex(
  pathname: string,
  options?: O.Assign<
    Omit<LoadIndexOptions, 'simplified'>,
    [{ simplified?: false }]
  >,
): Promise<IndexMap>;
export async function loadIndex(
  pathname: string,
  options: O.Assign<
    Omit<LoadIndexOptions, 'simplified'>,
    [{ simplified: true }]
  >,
): Promise<SimplifiedIndexMap>;
```

除这些外，indexer.ts 中并无更多改动。

5. 下面介绍 lexical-vector.ts 中的内容。该文件负责生成 TF-IDF 向量矩阵（使用 DataFrame 表示）。

先定义一个辅助函数 getAllDocIDs，用于读取一个索引字典中的所有文档 ID. 尽管在本次实验中，可以直接使用 1-30 万作为全部文档 ID，但实际上存在极少量的文档不包含任何合法词汇（如全部是特殊符号和标点），因此这么做不是非常严谨（事实上，得到的索引字典中只有 299861 个文档 ID，剩余的 100 多个文档不包含任何合法词汇）。并且，为了通用性考虑，单独定义这样一个函数也是合适的。

```
/**
 * Get all document IDs from the document ID map.
 * @param indexMap The document ID map, where the key is the term
 * and the value is a map where the key is the document ID
 * and the value is the list of indexes (start by 0) of the term in the sentence.
 * @returns
 */
const getAllDocIDs = (indexMap: SimplifiedIndexMap): number[] => {
  // Get the maximum document ID.
  // `Math.max` is not used as it can cause Maximum call stack size exceeded
  // when the number of sentences is too large.
  const elements = [...indexMap.values()]
    .map((m) => [...m.keys()])
    .flat()
    .map(Number);
  let maxElement = 0;
  for (const element of elements) {
    if (element > maxElement) {
      maxElement = element;
    }
  }

  const bitmap = createBitmap(maxElement);

  for (const docIDs of [...indexMap.values()].map((m) => [...m.keys()])) {
    for (const docID of docIDs.map(Number)) {
      bitmap.set(docID);
    }
  }
  return R.range(0, maxElement + 1).filter((docID) => bitmap.isSet(docID));
};
```

注意到这里使用了上次实验中在 `bitmap.ts` 中定义的位图数据结构。

然后定义生成 TF-IDF 向量矩阵的函数 `generateTFIDFVectorMatrix`。

```
/**
 * Generate a TF-IDF vector matrix from the document ID map.
 * @param indexMap The document ID map, where the key is the term
 * and the value is a map where the key is the document ID
 * and the value is the list of indexes (start by 0) of the term in the sentence.
 * @returns
 */
export const generateTFIDFVectorMatrix = (indexMap: SimplifiedIndexMap) => {
  const docIDs = getAllDocIDs(indexMap);
  const words = [...indexMap.keys()];

  const df = createSparseDataFrame<number>({
    shape: [words.length, docIDs.length],
    columnNames: docIDs.map(String),
    rowNames: words,
  });

  const docTotal = docIDs.length;
  for (const docID of docIDs) {
    for (const word of words) {
      const docID2wordCount = indexMap.get(word)!;
      const wordCount = docID2wordCount.get(docID);

      if (!wordCount) continue;

      const tf = 1 + Math.log(wordCount) / Math.log(10);

      const docCount = docID2wordCount.size;
      const idf = Math.log(docTotal / docCount) / Math.log(10);

      df.col[docID.toString()]!.at[word] = tf * idf;
    }

    if (docID % 300 === 0) {
      console.log(`TF-IDF Generated ${docID} / ${docTotal}`);
    }
  }

  return df;
};
```

注意这里必须使用稀疏矩阵（在这里就是 `SparseDataFrame`，其大致实现为通过一个对象（可以理解为 Python 中的字典）仅存储必要的值以避免占用过大的内存空间，具体实现将在附录中给出），因为若使用紧凑矩阵，其中将包含太多的 0.

在本次实验中，该矩阵的行数（词汇数）即使在词形还原后仍高达 43526 个，列数（文档数量）则高达 299861 个，按 Float32 存储，就需要占用 43526 × 299861 × 4=52206999544 字节=50983398KB=49788MB=48.62GB，这显然是没法塞进大多数家用电脑及笔记本的内存的（并且 Node.js 实际上仅支持分配最大占用内存 1.4G 的对象）。因此需要使用如稀疏矩阵等方式降低内存占用。

在克服了内存占用的问题后，关于该函数的实现思路则比较简单。它遍历文档 ID 及单词，获取索引字典中对应单词的 `docID2wordCount`，即一个嵌套的、键为文档 ID、值为该单词在该文档中的数量的字典。然后使用 `docID2wordCount.get(docID)`获取该单词在该文档中的数量、以此计算 TF，并使用该字典的长度 `docID2wordCount.size`（即该单词出现在了多少个文档中）计算 IDF，最后将 TF 与 IDF 的乘积作为 TF-IDF 值，并将其存储到稀疏矩阵的相应位置。

6. 最后在 `main.ts` 中编写主函数即可。

首先是一些常量及日志函数定义。这部分代码虽然较多但很容易理解，就不多解释了。

```
import fs from 'node:fs/promises';

import {
  generateIndex,
  lemmatize,
  loadIndex,
  saveIndex,
  transformIndexMap,
} from './indexer.js';
import { generateTFIDFVectorMatrix } from './lexical-vector.js';
import { loadSparseDataFrame } from './utils/dataframe.js';
import { fileExists } from './utils/file.js';
import { logged } from './utils/log.js';

import type { SparseDataFrame } from './utils/dataframe.js';

const CORPUS_PATHNAME = './data/en(utf8).txt';
const INDEX_PATHNAME = './dict.index';
const TFIDF_MATRIX_PATHNAME = './output/tfidf-matrix.csv';

const loggedReadCorpus = logged({
  message: 'Corpus read',
  fn: fs.readFile,
});
const loggedLoadIndex = logged({
  message: 'Index loaded',
  fn: loadIndex,
});
```

```
const loggedGenerateIndex = logged({
  message: 'Index generated',
  fn: generateIndex,
});
const loggedSaveIndex = logged({
  message: 'Index saved',
  fn: saveIndex,
  depth: 1,
});
const loggedTransformIndexMap = logged({
  message: 'Index map transformed',
  fn: transformIndexMap,
});
const loggedLoadTFIDFVectorMatrix = logged({
  message: 'TF-IDF matrix loaded',
  fn: loadSparseDataFrame<number>,
});
const loggedGenerateTFIDFVectorMatrix = logged({
  message: 'TF-IDF vector matrix generated',
  fn: generateTFIDFVectorMatrix,
});
const loggedSaveTFIDFVectorMatrix = logged({
  message: 'TF-IDF vector matrix saved',
  fn: (df: SparseDataFrame<number>) => ({
    toFile: (async (pathname) =>
      await df.saveToFile(pathname)) as (typeof df)['saveToFile'],
  }),
  depth: 1,
});
const loggedReadDocuments = logged({
  message: 'Documents read',
  fn: async (pathname: string) => {
    const content = await fs.readFile(pathname, 'utf-8');
    return content.split('\n');
  },
});
```

然后在主函数中，首先读取索引字典（用于计算待搜索文档的 TF-IDF 向量）和 TF-IDF 向量矩阵（用于计算待搜索文档和文档库中文档的相似度）。

```
const main = async () => {
  const indexMap = (await fileExists(INDEX_PATHNAME))
    ? await loggedLoadIndex(INDEX_PATHNAME, { simplified: true })
    : await (async () => {
        const corpus = await loggedReadCorpus(CORPUS_PATHNAME, 'utf-8');
        const documents = corpus.split('\n');
        const indexMap = await loggedGenerateIndex(documents);
        await loggedSaveIndex(indexMap).toFile(INDEX_PATHNAME);
        return loggedTransformIndexMap(indexMap);
      })();

  const df = (await fileExists(TFIDF_MATRIX_PATHNAME))
    ? await loggedLoadTFIDFVectorMatrix(TFIDF_MATRIX_PATHNAME)
    : await (async () => {
        const df = loggedGenerateTFIDFVectorMatrix(indexMap);
        await loggedSaveTFIDFVectorMatrix(df).toFile(TFIDF_MATRIX_PATHNAME);
        return df;
      })();

  console.log('\nTF-IDF vector matrix:');
  df.showHead();
  console.log();


  ...
}
```

可以看到，它们首先尝试读取已经保存的索引字典及 TF-IDF 向量矩阵，若不存在，则生成并保存，然后使用生成的对象。读取完毕后，会打印 TF-IDF 向量矩阵的部分内容进行展示。

接下来定义一个 `calculateSimilarities` 函数，它接受一个文档（待搜索的文档内容），将该文档分词并作词形还原后，计算其 TF-IDF 向量（这与 `generateTFIDFVectorMatrix` 中的实现非常类似）。同样的，这里也没有使用一个完整的紧凑数组，而是使用一个对象（可以理解为 Python 中的字典）表示其 TF-IDF 向量（即 `searchVec`）。然后，将该向量依次与 TF-IDF 矩阵中每个文档的 TF-IDF 向量归一化后点乘，得到文档相似度。

最后返回一个数组，其每一项是一个元组，第一项为文档 ID，第二项为待搜索文档与该文档的相似度。

```
const main = async () => {
  ...

  const calculateSimilarities = (document: string): Array<[number, number]> => {
    const words = document
      .split(' ')
      .filter((w) => /^[a-z]+$/.test(w))
      .map(lemmatize);
    const wordMap = new Map<string, number>();
    for (const word of words) {
      if (!wordMap.has(word)) wordMap.set(word, 0);
      wordMap.set(word, wordMap.get(word)! + 1);
    }

    const searchVec = df.rowNames.reduce((acc, word, index) => {
      const wordCount = wordMap.get(word);
      if (!wordCount) return acc;

      const tf = 1 + Math.log(wordCount) / Math.log(10);

      const docCount = indexMap.get(word)!.size;
      const idf = Math.log(df.columnNames.length / docCount) / Math.log(10);

      acc[index] = tf * idf;
      return acc;
    }, {} as Record<number, number>);

    const searchVecLen = Math.sqrt(
      Object.values(searchVec).reduce((acc, value) => acc + value ** 2, 0),
    );

    return df.columnNames.map((docID) => {
      const vec = df.col[docID]!;

      const vecLen = Math.sqrt(
        vec.accumulate((acc, value) => acc + value ** 2, 0),
      );

      const dotProd = Object.entries(searchVec).reduce(
        (acc, [index, value]) => acc + value * (vec.iat[Number(index)] ?? 0),
        0,
      );

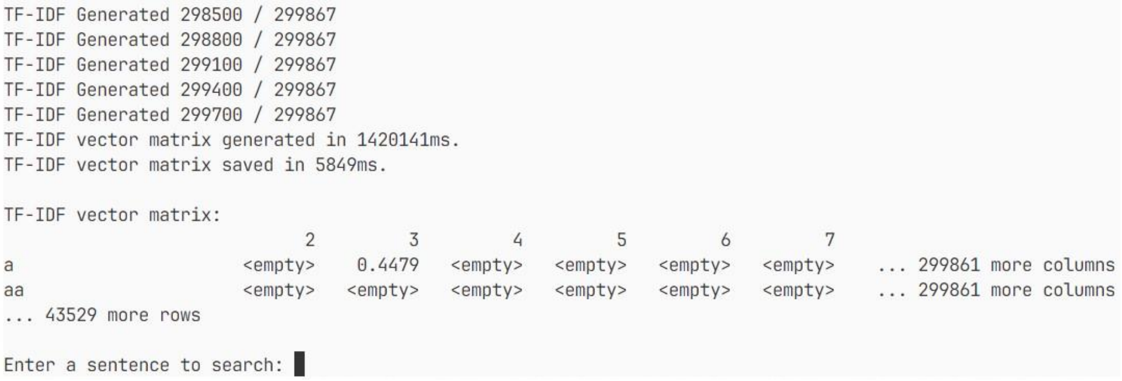
      return [Number(docID), dotProd / (searchVecLen * vecLen)];
    });
  };

  ...
}
```

注意到这里似乎自然而然地解决了两个向量长度可能不等的问题。其原因是两向量（待搜索文档的 TF-IDF 向量与待比较文档的 TF-IDF 向量）实际上均保留了标签信息（可以理解为 Pandas 中一个带索引的 Series）。在这里，待搜索文档的 TF-IDF 向量（即 `searchVec`）以对象形式表示，键为单词在单词序列中的索引（index），值为对应的 TF-IDF 值；而待比较文档的 TF-IDF 向量则直接是一个 `Series`，包含了完整的单词标签。

因此，可以直接将它们的值依此相乘最后相加得到点乘结果，如某个单词的对应值为空，则将其作为 0 处理。可以理解为仅将两个向量中都存在的单词对应的值相乘（交集），这样不仅表示很简单，速度也很快。

最后，再编写读取用户输入并打印搜索结果的代码即可：

```
const main = async () => {
  ...
  const documents = await loggedReadDocuments(CORPUS_PATHNAME);

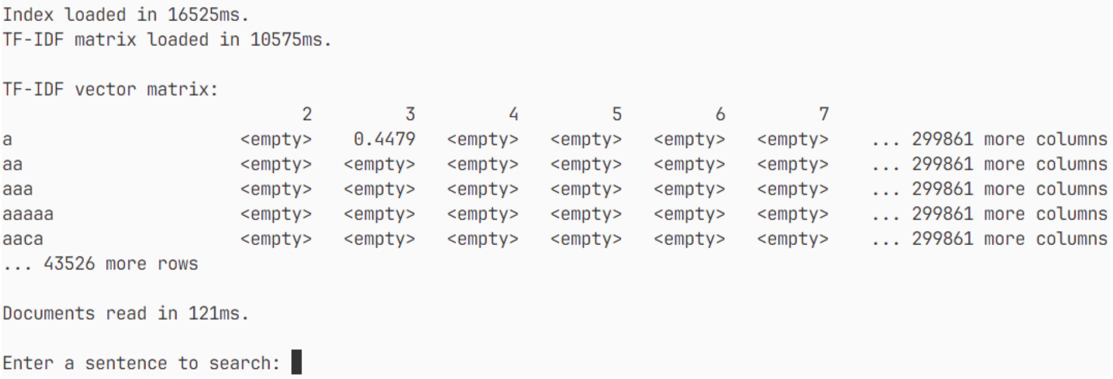
  process.stdout.write('\nEnter a sentence to search: ');
  process.stdin.on('data', async (data) => {
    const similarities = logged({
      message: ({ __callStack: [[, [input]]] }) => `Queried "${input}"`,
      fn: calculateSimilarities,
    })(data.toString().trim());
    const similaritiesToShow = similarities
      .sort((a, b) => b[1] - a[1])
      .slice(0, 5);

    similaritiesToShow.forEach(([docID, score], index) => {
      console.log(
        `${index + 1}. Similarity: ${score}\n` +
          `   Document ID: ${docID}\n` +
          `   Document: ${documents[docID] - 1}`,
      );
    });

    process.stdout.write('\nEnter a sentence to search: ');
  });
};

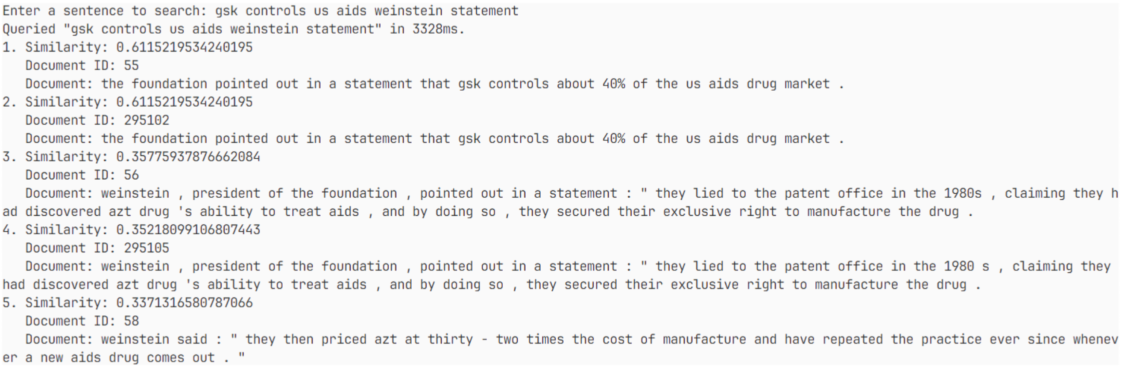
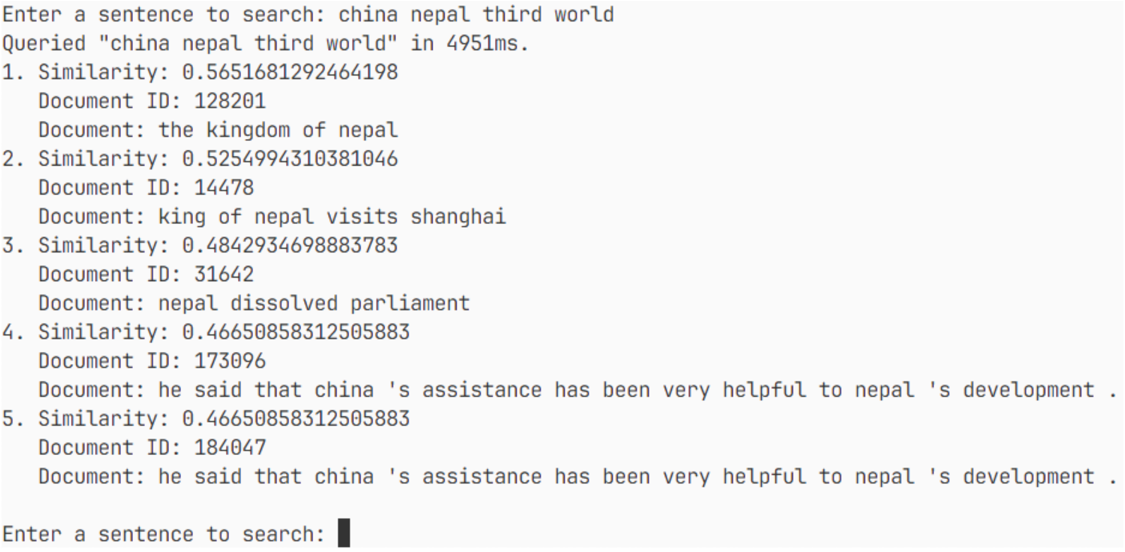
await main();
```

这部分代码很简单，其循环读取用户输入的待搜索文档，通过 `calculateSimilarities` 计算其与所有文档的相似度，从大到小排序后取前五个分别打印相似度、文档 ID 及文档内容。

（二）实验结果

运行 `npm run dev`，使用 ts-node 执行`./src/main.ts`。下面展示输出结果。

首先是第一次运行时计算 TF-IDF 向量矩阵的输出，可以看到大约耗费了 1400s：



```
TF-IDF Generated 298500 / 299867
TF-IDF Generated 298800 / 299867
TF-IDF Generated 299100 / 299867
TF-IDF Generated 299400 / 299867
TF-IDF Generated 299700 / 299867
TF-IDF vector matrix generated in 1420141ms.
TF-IDF vector matrix saved in 5849ms.

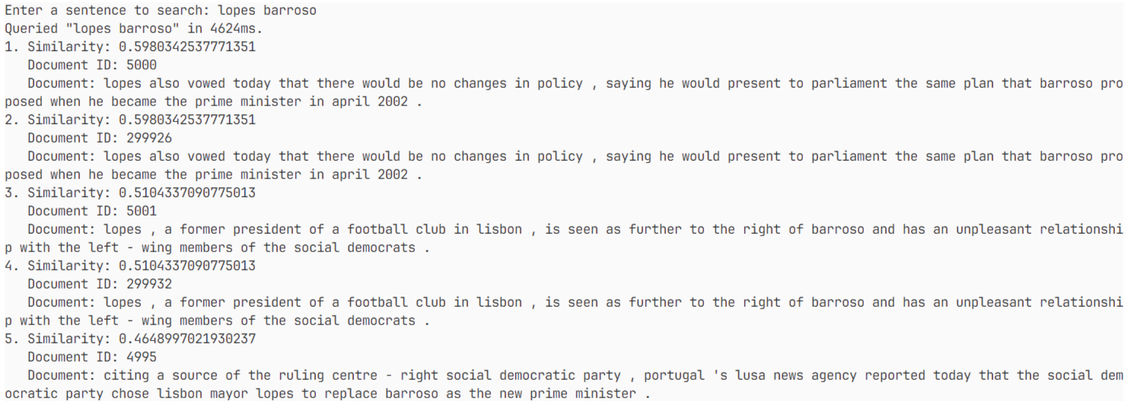
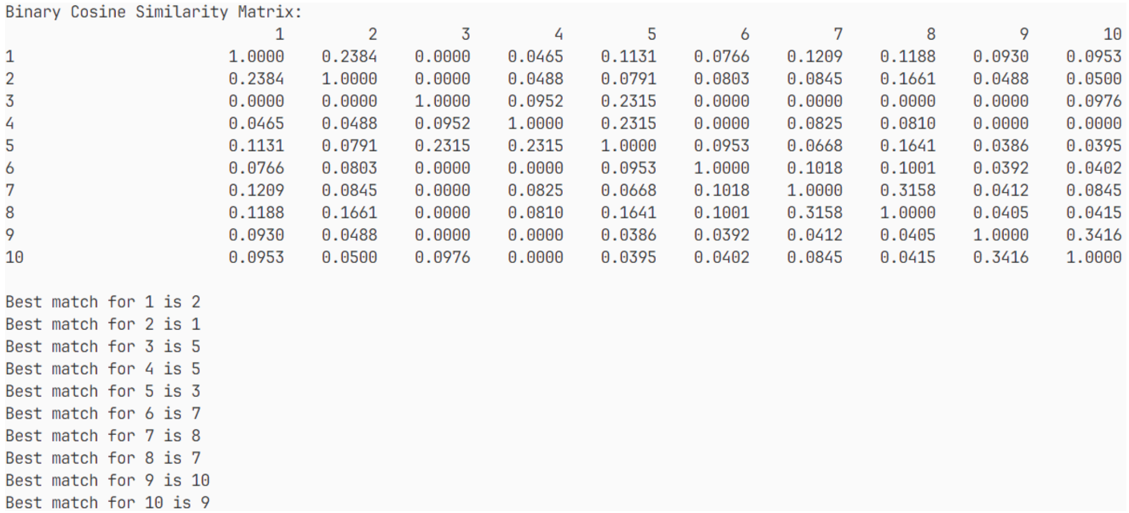
TF-IDF vector matrix:
                             2         3         4         5         6         7
a                      <empty>    0.4479   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
aa                     <empty>   <empty>   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
... 43529 more rows

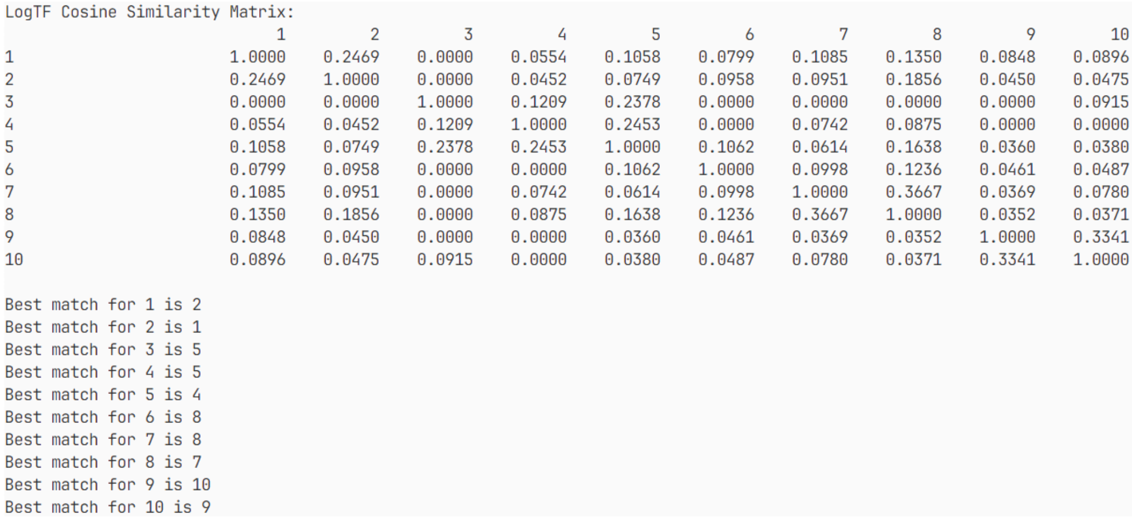
Enter a sentence to search: █
```

图 2 第一次运行时计算 TF-IDF 矩阵的输出

下为第二次及之后运行时的输出：



```
Index loaded in 16525ms.
TF-IDF matrix loaded in 10575ms.

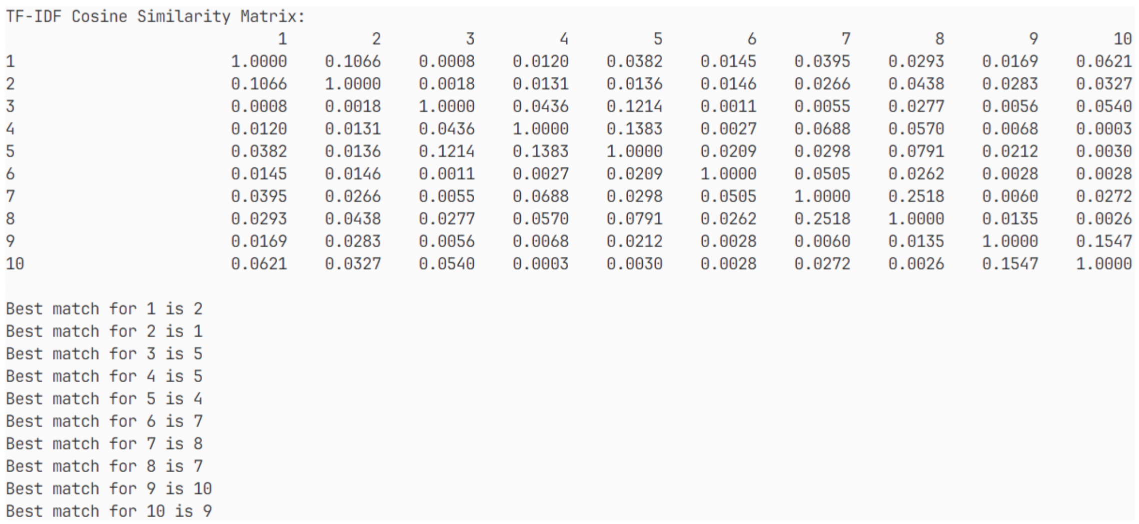
TF-IDF vector matrix:
                             2         3         4         5         6         7
a                      <empty>    0.4479   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
aa                     <empty>   <empty>   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
aaa                    <empty>   <empty>   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
aaaaa                  <empty>   <empty>   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
aaca                   <empty>   <empty>   <empty>   <empty>   <empty>   <empty>    ... 299861 more columns
... 43526 more rows

Documents read in 121ms.

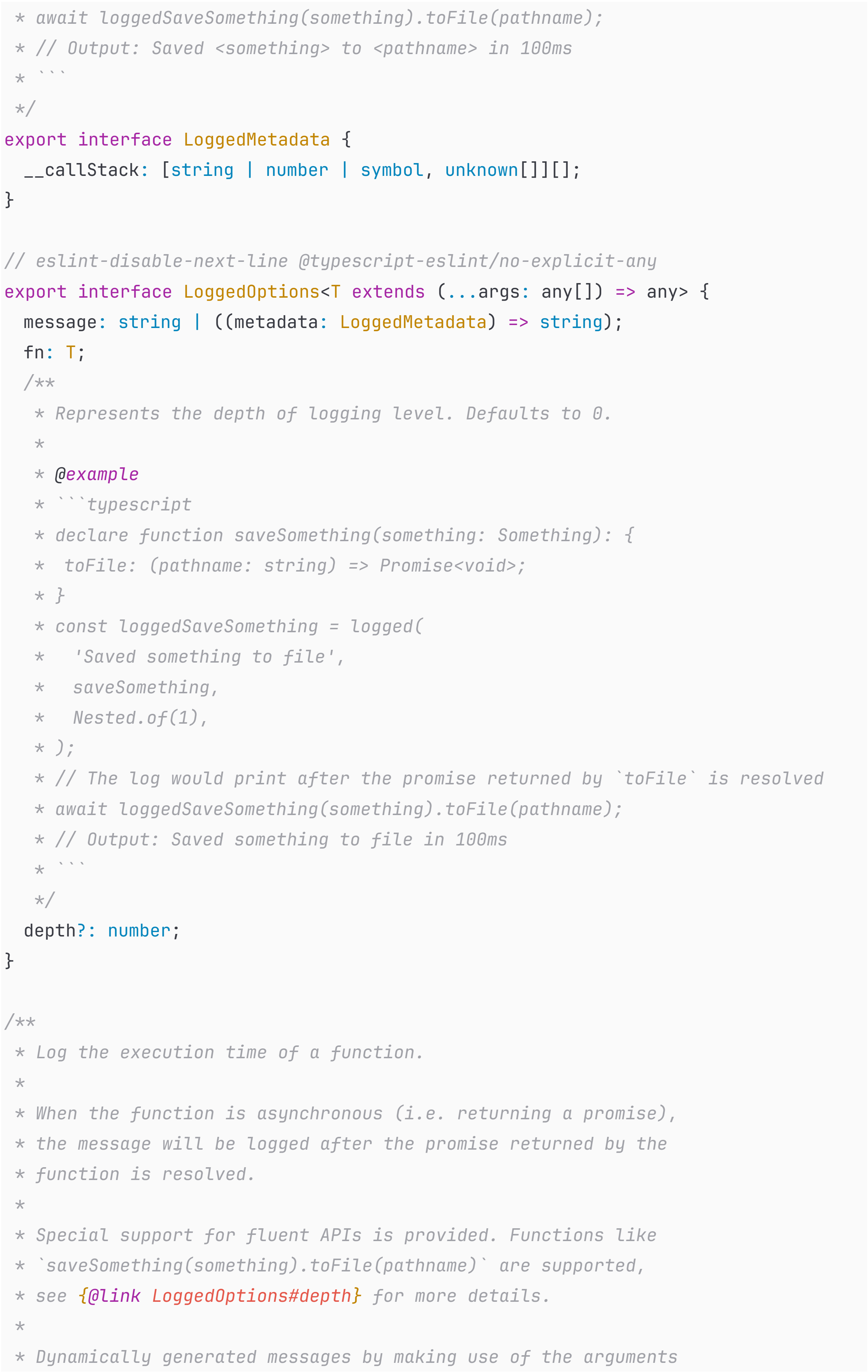
Enter a sentence to search: █
```

图 3 第二次及之后运行时的输出

然后展示题中所给的对三条查询的搜索结果：



```
Enter a sentence to search: gsk controls us aids weinstein statement
Queried "gsk controls us aids weinstein statement" in 3328ms.
1. Similarity: 0.6115219534240195
   Document ID: 55
   Document: the foundation pointed out in a statement that gsk controls about 40% of the us aids drug market .
2. Similarity: 0.6115219534240195
   Document ID: 295102
   Document: the foundation pointed out in a statement that gsk controls about 40% of the us aids drug market .
3. Similarity: 0.35775937876662084
   Document ID: 56
   Document: weinstein , president of the foundation , pointed out in a statement : " they lied to the patent office in the 1980s , claiming they h
ad discovered azt drug 's ability to treat aids , and by doing so , they secured their exclusive right to manufacture the drug .
4. Similarity: 0.35218099106807443
   Document ID: 295105
   Document: weinstein , president of the foundation , pointed out in a statement : " they lied to the patent office in the 1980 s , claiming they
had discovered azt drug 's ability to treat aids , and by doing so , they secured their exclusive right to manufacture the drug .
5. Similarity: 0.3371316580787066
   Document ID: 58
   Document: weinstein said : " they then priced azt at thirty - two times the cost of manufacture and have repeated the practice ever since whenev
er a new aids drug comes out . "
```

图 4 查询结果 1

可以看到第一条查询的搜索结果按相似度从高到低排序，文档 ID 分别是 55、295102、60、295099 和 61.



```
Enter a sentence to search: china nepal third world
Queried "china nepal third world" in 4951ms.
1. Similarity: 0.5651681292464198
   Document ID: 128201
   Document: the kingdom of nepal
2. Similarity: 0.5254994310381046
   Document ID: 14478
   Document: king of nepal visits shanghai
3. Similarity: 0.4842934698883783
   Document ID: 31642
   Document: nepal dissolved parliament
4. Similarity: 0.46650858312505883
   Document ID: 173096
   Document: he said that china 's assistance has been very helpful to nepal 's development .
5. Similarity: 0.46650858312505883
   Document ID: 184047
   Document: he said that china 's assistance has been very helpful to nepal 's development .

Enter a sentence to search: █
```

图 5 查询结果 2

第二条查询的搜索结果按相似度从高到低排序，文档 ID 分别是 128201、14478、31642、173096 和 184047.



```
Enter a sentence to search: lopes barroso
Queried "lopes barroso" in 4624ms.
1. Similarity: 0.5980342537771351
   Document ID: 5000
   Document: lopes also vowed today that there would be no changes in policy , saying he would present to parliament the same plan that barroso pro
posed when he became the prime minister in april 2002 .
2. Similarity: 0.5980342537771351
   Document ID: 299926
   Document: lopes also vowed today that there would be no changes in policy , saying he would present to parliament the same plan that barroso pro
posed when he became the prime minister in april 2002 .
3. Similarity: 0.5104337090775013
   Document ID: 5001
   Document: lopes , a former president of a football club in lisbon , is seen as further to the right of barroso and has an unpleasant relationshi
p with the left - wing members of the social democrats .
4. Similarity: 0.5104337090775013
   Document ID: 299932
   Document: lopes , a former president of a football club in lisbon , is seen as further to the right of barroso and has an unpleasant relationshi
p with the left - wing members of the social democrats .
5. Similarity: 0.4648997021930237
   Document ID: 4995
   Document: citing a source of the ruling centre - right social democratic party , portugal 's lusa news agency reported today that the social dem
ocratic party chose lisbon mayor lopes to replace barroso as the new prime minister .
```

图 6 查询结果 3

第三条查询的搜索结果按相似度从高到低排序，文档 ID 分别是 5000、299926、5001、299932 和 4995.



```
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 7 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 8 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 9 TF-IDF 计算结果

（附）log.ts、series.ts 及 dataframe.ts 的代码实现

```
// log.ts
/**
 * The metadata of a logged function.
 *
 * @example
 * ```typescript
 * declare function queryAndPrint(input: string): void;
 * const loggedQueryAndPrint = logged({
 *   message: ({ __callStack: [[, [input]]] }) => `Queried "${input}"`,
 *   fn: queryAndPrint,
 * });
 * // The `__callStack` property is `[['queryAndPrint', ['Hello, world!']]]` here
 * loggedQueryAndPrint('Hello, world!');
 * // Output: Queried "Hello, world!" in 100ms
 *
 * declare function saveSomething(something: Something): {
 *   toFile: (pathname: string) => Promise<void>;
 * }
 * const loggedSaveSomething = logged({
 *   message: ({ __callStack: [[, [something]], [, [pathname]]] }) =>
 *     `Saved ${something} to ${pathname}`,
 *   fn: saveSomething,
 *   depth: 1,
 * });
 * // The `__callStack` property is `[
 * //   ['saveSomething', [something]],
 * //   ['toFile', [pathname]],
 * // ]` here
```



```typescript
 * await loggedSaveSomething(something).toFile(pathname);
 * // Output: Saved <something> to <pathname> in 100ms
 * ```
 */
export interface LoggedMetadata {
  __callStack: [string | number | symbol, unknown[]][];
}

// eslint-disable-next-line @typescript-eslint/no-explicit-any
export interface LoggedOptions<T extends (...args: any[]) => any> {
  message: string | ((metadata: LoggedMetadata) => string);
  fn: T;
  /**
   * Represents the depth of logging level. Defaults to 0.
   *
   * @example
   * ```typescript
   * declare function saveSomething(something: Something): {
   *  toFile: (pathname: string) => Promise<void>;
   * }
   * const loggedSaveSomething = logged(
   *   'Saved something to file',
   *   saveSomething,
   *   Nested.of(1),
   * );
   * // The log would print after the promise returned by `toFile` is resolved
   * await loggedSaveSomething(something).toFile(pathname);
   * // Output: Saved something to file in 100ms
   * ```
   */
  depth?: number;
}

/**
 * Log the execution time of a function.
 *
 * When the function is asynchronous (i.e. returning a promise),
 * the message will be logged after the promise returned by the
 * function is resolved.
 *
 * Special support for fluent APIs is provided. Functions like
 * `saveSomething(something).toFile(pathname)` are supported,
 * see {@link LoggedOptions#depth} for more details.
 *
 * Dynamically generated messages by making use of the arguments
```

```
 * passed to the function are also supported. See {@link LoggedMetadata}.
 *
 * A basic example with static message and no fluent API is shown below
 * (if you want to learn more, check {@link LoggedMetadata} and {@link LoggedOptions#depth}).
 * @param options The options for logging.
 *
 * @example
 * ```typescript
 * declare function queryAndPrint(input: string): void;
 * const loggedQueryAndPrint = logged({
 *   message: 'Queried something',
 *   fn: queryAndPrint,
 * });
 * // The log would print after the function returns
 * loggedQueryAndPrint('Hello, world!');
 * // Output: Queried something in 100ms
 *
 * declare function save(file: File, pathname: string): Promise<void>;
 * const loggedSave = logged({ message: 'File saved', fn: save });
 * // The log would print after the promise returned by `save` is resolved
 * await loggedSave(file, pathname);
 * // Output: File saved in 100ms
 * ```
 */
// eslint-disable-next-line @typescript-eslint/no-explicit-any
export const logged = <T extends (...args: any[]) => any>(
  options: LoggedOptions<T>,
): T => {
  const { depth = 0, fn, message } = options;

  // WARNING: Many confusing methods are used here in order to
  // make accessing the metadata possible, so if you are not interested
  // in the implementation, just ignore it.
  return ((...args) => {
    const startTime = Date.now();
    let result = fn(...args);

    // Extract metadata from the last call.
    let metadata: LoggedMetadata;
    if (
      result !== null &&
      typeof result === 'object' &&
      '__metadata' in result
    ) {
```

```
    metadata = result['__metadata'];
    result = result['result'];
  } else {
    // If no metadata is found, consider this function as the root function,
    // and create a new metadata.
    metadata = { __callStack: [[fn.name, args]] };
  }

  // For async functions, wait for the result and log the message.
  if (result instanceof Promise) {
    result = result
      .then((res) => {
        if (depth === 0) {
          console.log(
            `${
              typeof message === 'string' ? message : message(metadata ?? [])
            } in ${Date.now() - startTime}ms.`,
          );
        }
        return res;
      })
      .catch((err) => {
        throw err;
      });
  }

  if (!(result instanceof Promise) && depth === 0) {
    // If the depth is 0, log the message
    console.log(
      `${
        typeof message === 'string' ? message : message(metadata ?? [])
      } in ${Date.now() - startTime}ms.`,
    );
  }

  if (depth === 0) return result;

  // Check if the result is an object if the nested level is not 0.
  if (typeof result !== 'object' || result === null) {
    throw new Error('Nested logging is only supported for objects.');
  }

  // Recursively log the nested functions.
  return Object.entries(result).reduce((acc, [key, value]) => {
    acc[key] = value;
```

```
      if (typeof value === 'function') {
        acc[key] = logged({
          message,
          // eslint-disable-next-line @typescript-eslint/no-explicit-any
          fn: (...as: any[]) => ({
            __metadata: {
              ...metadata,
              __callStack: [...metadata.__callStack, [key, as]],
            } as LoggedMetadata,
            result: value(...as),
          }),
          depth: depth - 1,
        });
      }
      return acc;
      // eslint-disable-next-line @typescript-eslint/no-explicit-any
    }, {} as any);
  }) as T;
};
```

```
// series.ts
import * as R from 'ramda';

import type { Tuple as T, Union as U, Number as N } from 'ts-toolbelt';

/**
 * A series is a one-dimensional array with labels.
 * @template TData The type of the data.
 * @template TLabels The type of the labels, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `at`
 * to `TData | undefined`.
 * @template TType The type of the series, either `'normal'` or `'sparse'`.
 * A sparse series internally uses a `Record<number, TData>` (an object)
 * to store data, while a normal series uses a `TData[]` (an array).
 *
 * @see {@link createSeries}
 */
export interface Series<
  TData,
  TLabels extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
> {
```

```
/**
 * The type of the series, either `'normal'` or `'sparse'`.
 */
readonly type: TType;
/**
 * The length of the series.
 *
 * If `TLabels` is not the fallback type `string[]`,
 * then the type of it would be the exact length of the labels (literal number).
 * Otherwise, it would be just `number`.
 *
 * It is a O(1) operation, whether the series is sparse or not.
 */
readonly length: TLabels['length'];
/**
 * The labels of the series.
 */
readonly labels: TLabels;
/**
 * The internal data of the series.
 *
 * If the series is sparse, then it would be a `Record<number, TData>` (an object),
 * otherwise it would be a `TData[]` (an array).
 *
 * It is not recommended to access this property directly,
 * unless you know what you are doing.
 */
readonly $data: TType extends 'normal' ? TData[] : Record<number, TData>;

/**
 * Set all values of the series.
 * @param values Either an array of values, or a series. In case of sparse series,
 * the array can contain `undefined` to indicate missing values.
 *
 * @example
 * ```typescript
 * const ser1 = createSeries({
 *   data: [1, 2, 3, 4, 5],
 *   labels: ['a', 'b', 'c', 'd', 'e'],
 * });
 * ser1.$setAll([42, 43, 44, 45, 46]); // `ser1` is now [42, 43, 44, 45, 46]
 * const ser2 = createSeries({
 *  data: [99, 100],
 *  labels: ['b', 'e'],
 * });
```

```
 * ser1.$setAll(ser2); // `ser1` is now [42, 99, 44, 45, 100]
 *
 * const sparseSer = createSeries({
 *   type: 'sparse',
 *   length: 3,
 *   labels: ['a', 'b', 'c'],
 * });
 * sparseSer.$setAll([1, undefined, 3]); // `sparseSer` is now [1, <empty>, 3]
 * ```
 */
$setAll(
  values:
    | (TType extends 'normal' ? TData[] : Array<TData | undefined>)
    | Series<TData, TLabels, 'normal' | 'sparse'>,
): void;

/**
 * Get or set the value at the given label (not index).
 *
 * **Typesafety is ensured** when `TLabels` is the fallback type `string[]`,
 * i.e. the type of the return value is `TData | undefined`.
 * While in exact type, the type of the return value is `TData`.
 *
 * @example
 * ```typescript
 * const ser = createSeries({
 *   data: [1, 2, 3, 4, 5],
 *   labels: ['a', 'b', 'c', 'd', 'e'],
 * });
 * ser.at['b']; // 2
 * ser.at['b'] = 42; // `ser` is now [1, 42, 3, 4, 5]
 * ser.at['b']; // 42
 * ```
 */
at: {
  [Label in TLabels[number]]: string[] extends TLabels
    ? TData | undefined
    : TData;
};
/**
 * Get or set the value at the given integer position, i.e. index (not label).
 *
 * Typesafety is **not** ensured when `TLabels` is the fallback type `string[]`,
 * so the type of the return value is always `TData`.
 * You have to ensure the index is valid by yourself.
```

```
 *
 * @example
 * ```typescript
 * const ser = createSeries({
 *   data: [1, 2, 3, 4, 5],
 *   labels: ['a', 'b', 'c', 'd', 'e'],
 * });
 * ser.iat[1]; // 2
 * ser.iat[1] = 42; // `ser` is now [1, 42, 3, 4, 5]
 * ser.iat[1]; // 42
 * ```
 */
iat: { [key: number]: TData };

/**
 * Convert the series to an array.
 *
 * If the series is sparse, then the array would contain `undefined`
 * to indicate missing values.
 * @returns The array representation of the series.
 *
 * @example
 * ```typescript
 * const ser = createSeries({
 *   data: [1, 2, 3, 4, 5],
 *   labels: ['a', 'b', 'c', 'd', 'e'],
 * });
 * ser.toArray(); // [1, 2, 3, 4, 5]
 *
 * const sparseSer = createSeries({
 *  type: 'sparse',
 *  length: 3,
 *  labels: ['a', 'b', 'c'],
 * });
 * sparseSer.toArray(); // [undefined, undefined, undefined]
 * sparseSer['b'] = 42;
 * sparseSer.toArray(); // [undefined, 42, undefined]
 * ```
 */
toArray(): TType extends 'normal' ? TData[] : Array<TData | undefined>;

/**
 * Clone the series.
 * @returns A new series with the same data and labels.
 */
```

```typescript
  clone(): Series<TData, TLabels, TType>;

  /**
   * Create a new series with the given labels.
   * @param labels The new labels.
   * @returns A new series with the same data and the given labels.
   */
  withLabels<const NTLabels extends readonly string[]>(
    labels: NTLabels,
  ): NTLabels['length'] extends TLabels['length']
    ? Series<TData, NTLabels, TType>
    : never;

  /**
   * Transform the series by applying the given transformer to each value,
   * similar to `Array#map`.
   * @param transformer The callback function to transform each value.
   *
   * @example
   * ```typescript
   * const ser = createSeries({
   *   data: [1, 2, 3, 4, 5],
   *   labels: ['a', 'b', 'c', 'd', 'e'],
   * });
   * const newSer = ser.transform((value) => value * 2); // [2, 4, 6, 8, 10]
   *
   * const sparseSer = createSeries({
   *   type: 'sparse',
   *   length: 3,
   *   labels: ['a', 'b', 'c'],
   * });
   * sparseSer['a'] = 1;
   * sparseSer['c'] = 3;
   * const newSparseSer = sparseSer.transform((value) => value * 2); // [2, <empty>,
6]
   * ```
   */
  transform<NTData>(
    transformer: (
      value: TData,
      index?: number,
      label?: TLabels[number],
      series?: Series<TData, TLabels, TType>,
    ) => NTData,
  ): Series<NTData, TLabels, TType>;
```

```
/**
 * Accumulate the series by applying the given accumulator to each value,
 * similar to `Array#reduce`, but `initialValue` can be omitted by using
 * the first value of the series as the initial value.
 * @param accumulator The callback function to accumulate each value.
 * @param initialValue The initial value of the accumulator. If omitted,
 * the first value of the series would be used as the initial value.
 * @returns The accumulated value.
 */
accumulate<NTData>(
  accumulator: (
    acc: NTData,
    value: TData,
    index?: number,
    label?: TLabels[number],
    series?: Series<TData, TLabels, TType>,
  ) => NTData,
  initialValue: NTData,
): NTData;
accumulate(
  accumulator: (
    acc: TData,
    value: TData,
    index?: number,
    label?: TLabels[number],
    series?: Series<TData, TLabels, TType>,
  ) => TData,
): TData;
/**
 * Similar to `Series#accumulate`, but the values are accumulated from right to left.
 * @param accumulator The callback function to accumulate each value.
 * @param initialValue The initial value of the accumulator. If omitted,
 * the last value of the series would be used as the initial value.
 * @returns The accumulated value.
 *
 * @see {@link Series#accumulate}
 */
accumulateRight<NTData>(
  accumulator: (
    acc: NTData,
    value: TData,
    index?: number,
    label?: TLabels[number],
    series?: Series<TData, TLabels, TType>,
  ) => NTData,
    initialValue: NTData,
  ): NTData;
  accumulateRight(
    accumulator: (
      acc: TData,
      value: TData,
      index?: number,
      label?: TLabels[number],
      series?: Series<TData, TLabels, TType>,
    ) => TData,
  ): TData;

  [Symbol.toStringTag]: 'Series';
}
/**
 * The sparse version of `Series`.
 * It is just the same to `Series<TData, TLabels, 'sparse'>`.
 */
export type SparseSeries<
  TData,
  TLabels extends readonly string[] = string[],
> = Series<TData, TLabels, 'sparse'>;

interface CreateSeriesCommonOptions<TLabels extends readonly string[]> {
  $checkLabels?: boolean;
  $copyLabels?: boolean;
  $label2indexMap?: Record<TLabels[number], number>;
  labels?: TLabels;
}
interface CreateSeriesVectorNormalOptions<TData> {
  data: TData[];
}
interface CreateSeriesVectorSparseOptions {
  length: number;
}
type CreateSeriesVectorOptions<TData, TType extends 'normal' | 'sparse'> = {
  type?: TType;
} & (TType extends 'normal'
  ? CreateSeriesVectorNormalOptions<TData>
  : CreateSeriesVectorSparseOptions);
/**
 * The options for {@link createSeries}.
 *
 * @see {@link createSeries}
 */
```

```
export type CreateSeriesOptions<
  TData,
  TLabels extends readonly string[],
  TType extends 'normal' | 'sparse',
> = CreateSeriesCommonOptions<TLabels> &
  CreateSeriesVectorOptions<TData, TType>;
/**
 * The options for {@link createNormalSeries}.
 *
 * @see {@link createSparseSeries}
 */
export type CreateSparseSeriesOptions<TLabels extends readonly string[]> =
  CreateSeriesCommonOptions<TLabels> & CreateSeriesVectorSparseOptions;

/**
 * Create a series (normal or sparse) from the given options.
 * @param options The options to create the series.
 * @returns A series.
 *
 * @example
 * ```typescript
 * // Create a normal series (data are stored in an array)
 * const ser = createSeries({
 *   data: [1, 2, 3, 4, 5],
 *   labels: ['a', 'b', 'c', 'd', 'e'],
 * });
 *
 * // Create a sparse series (data are stored in an object)
 * // `data` cannot be provided in this case, instead `length` must be provided
 * // to specify the length of the series
 * // Note that in this case the data type (i.e. `TData`) cannot be inferred
 * // by TypeScript, so you have to specify it explicitly
 * // Also, the type of labels (i.e. `TLabels`) should be provided by your own
 * // if you want to keep it exact
 * const labels = ['a', 'b', 'c'] as const;
 * const sparseSer = createSeries<number, typeof labels>({
 *   type: 'sparse',
 *   length: 3,
 *   labels,
 * });
 * ```
 */
export const createSeries = <
  TData,
  const TLabels extends readonly string[] = string[],
```

```
  TType extends 'normal' | 'sparse' = 'normal',
>(
  options: CreateSeriesOptions<TData, TLabels, TType>,
): Series<TData, TLabels, TType> => {
  const {
    $checkLabels = true,
    $copyLabels = true,
    $label2indexMap,
    labels,
  } = options;

  if ($checkLabels && labels && R.uniq(labels).length !== labels.length)
    throw new Error('Labels must be unique');

  const _length =
    options.type === 'sparse'
      ? (options as CreateSeriesVectorSparseOptions).length
      : (options as CreateSeriesVectorNormalOptions<TData>).data.length;

  const _data =
    options.type === 'sparse'
      ? ({} as Record<number, TData>)
      : [...(options as CreateSeriesVectorNormalOptions<TData>).data];

  const _labels =
    labels !== undefined
      ? $copyLabels
        ? [...labels]
        : labels
      : R.range(0, _length).map(String);
  if (!Object.isFrozen(_labels)) Object.freeze(_labels);

  const _label2index =
    $label2indexMap ??
    (R.fromPairs(_labels.map((row, i) => [row, i])) as Record<
      TLabels[number],
      number
    >);

  const result: Series<TData, TLabels, TType> = {
    get type() {
      return (options.type ?? 'normal') as TType;
    },
    get length() {
      return _length;
    },
    get labels() {
      return _labels as unknown as TLabels;
    },
    get $data() {
      return _data as Series<TData, TLabels, TType>['$data'];
    },

    at: new Proxy(
      {},
      {
        get(_, label: TLabels[number]) {
          return _data[_label2index[label]];
        },
        set(_, label: TLabels[number], val) {
          _data[_label2index[label]] = val;
          return true;
        },
      },
    ) as Series<TData, TLabels, TType>['at'],
    iat: new Proxy(
      {},
      {
        get(_, index) {
          return _data[index as unknown as number];
        },
        set(_, index, val) {
          _data[index as unknown as number] = val;
          return true;
        },
      },
    ),

    $setAll(values) {
      if (options.type === 'sparse') {
        if (Array.isArray(values)) {
          values.forEach((value, index) => {
            if (value === undefined) return;
            _data[index] = value;
          });
        } else if (
          (values as Series<TData, TLabels, TType>).type === 'normal'
        ) {
          const other = values as Series<TData, TLabels, 'normal'>;
          other.labels.forEach((label) => {
```

```
        const index = _label2index[label as TLabels[number]];
        if (index === undefined) return;
        _data[index] = other.at[label as (typeof other.labels)[number]]!;
      });
    } else {
      const other = values as Series<TData, TLabels, 'sparse'>;
      other.labels.forEach((label) => {
        const index = _label2index[label as TLabels[number]];
        if (index === undefined) return;
        const val = other.at[label as (typeof other.labels)[number]];
        if (val === undefined) return;
        _data[index] = val!;
      });
    }

    return;
  }

  if (Array.isArray(values)) {
    (_data as TData[]).splice(
      0,
      (_data as TData[]).length,
      ...(values as TData[]),
    );
  } else if ((values as Series<TData, TLabels, TType>).type === 'normal') {
    const other = values as Series<TData, TLabels, 'normal'>;
    other.labels.forEach((label) => {
      const index = _label2index[label as TLabels[number]];
      if (index === undefined) return;
      _data[index] = other.at[label as (typeof other.labels)[number]]!;
    });
  } else {
    const other = values as Series<TData, TLabels, 'sparse'>;
    other.labels.forEach((label) => {
      const index = _label2index[label as TLabels[number]];
      if (index === undefined) return;
      const val = other.at[label as (typeof other.labels)[number]];
      if (val === undefined) return;
      _data[index] = val!;
    });
  }
},

toArray() {
  if (options.type === 'sparse')
```

```
      return Array.from({ length: _length }, (_, i) => _data[i]);

    return [...(_data as TData[])];
  },

  clone() {
    if (options.type === 'sparse') {
      const result = createSeries(options);
      for (const [index, value] of Object.entries(_data)) {
        const indexNumber = Number(index);
        result.$data[indexNumber] = value;
      }
      return result;
    }

    return createSeries<TData, TLabels, TType>({
      ...options,
      data: [...(_data as TData[])],
    });
  },

  withLabels(labels) {
    if (options.type === 'sparse') {
      const result = createSeries({ ...options, labels });
      for (const [index, value] of Object.entries(_data)) {
        const indexNumber = Number(index);
        result.$data[indexNumber] = value;
      }
      // eslint-disable-next-line @typescript-eslint/no-explicit-any
      return result as any;
    }

    return createSeries({
      ...options,
      labels,
      data: [...(_data as TData[])],
    });
  },

  transform(transformer) {
    if (options.type === 'sparse') {
      const newSer = createSeries(options);
      for (const [index, value] of Object.entries(_data)) {
        const indexNumber = Number(index);
        (newSer.$data[indexNumber] as unknown) = transformer(
          value,
          indexNumber,
          _labels[indexNumber],
          result,
          // eslint-disable-next-line @typescript-eslint/no-explicit-any
        ) as any;
      }
      return newSer;
    }

    const newData = (_data as TData[]).map((value, index) =>
      transformer(value, index, _labels[index], result),
    );
    return createSeries({
      ...options,
      data: newData,
      // eslint-disable-next-line @typescript-eslint/no-explicit-any
    }) as any;
  },
  // @ts-expect-error - overload
  accumulate(accumulator, initialValue) {
    let flag = initialValue === undefined;

    if (options.type === 'sparse') {
      let res = initialValue;
      for (const [index, value] of Object.entries(_data)) {
        if (flag) {
          (res as unknown) = value;
          flag = false;
          continue;
        }
        const indexNumber = Number(index);
        (res as unknown) = accumulator(
          // eslint-disable-next-line @typescript-eslint/no-explicit-any
          res as any,
          value,
          indexNumber,
          _labels[indexNumber],
          result,
        );
      }
      return res;
    }

    let res = initialValue;
    for (let i = 0; i < _length; i++) {
      if (flag) {
        (res as unknown) = (_data as TData[])[i];
        flag = false;
        continue;
      }
      (res as unknown) = accumulator(
        // eslint-disable-next-line @typescript-eslint/no-explicit-any
        res as any,
        (_data as TData[])[i],
        i,
        _labels[i],
        result,
      );
    }
    return res;
  },
  // @ts-expect-error - overload
  accumulateRight(accumulator, initialValue) {
    let flag = initialValue === undefined;

    if (options.type === 'sparse') {
      let res = initialValue;
      for (const index of Object.keys(_data).reverse()) {
        const indexNumber = index as unknown as number;
        const value = _data[indexNumber];
        if (flag) {
          (res as unknown) = value;
          flag = false;
          continue;
        }
        (res as unknown) = accumulator(
          // eslint-disable-next-line @typescript-eslint/no-explicit-any
          res as any,
          value,
          indexNumber,
          _labels[indexNumber],
          result,
        );
      }
      return res;
    }

    let res = initialValue;
    for (let i = _length - 1; i >= 0; i--) {
        if (flag) {
          (res as unknown) = (_data as TData[])[i];
          flag = false;
          continue;
        }
        (res as unknown) = accumulator(
          // eslint-disable-next-line @typescript-eslint/no-explicit-any
          res as any,
          (_data as TData[])[i],
          i,
          _labels[i],
          result,
        );
      }
      return res;
    },

    [Symbol.toStringTag]: 'Series',
  };

  return result;
};
/**
 * Creates a sparse series. It is just the same when passing
 * `type: 'sparse'` to `createSeries`.
 * @param options
 * @returns A sparse series.
 *
 * @see {@link createSeries}
 */
export const createSparseSeries = <
  TData,
  TLabels extends readonly string[] = string[],
>(
  options: CreateSparseSeriesOptions<TLabels>,
): SparseSeries<TData, TLabels> =>
  createSeries<TData, TLabels, 'sparse'>({ ...options, type: 'sparse' });

/**
 * Creates a normal series from an array of values, same as `createSeries({ data: values })`.
 * It is just another version of {@link createSeries} with less options. If you don't care
 * about labels, it is a cleaner way to create a series.
 *
```

```typescript
 * If you want to create a sparse one, see {@link sparseSeries}.
 * @param data The array of values.
 * @returns A normal series.
 *
 * @see {@link createSeries}
 *
 * @example
 * ```typescript
 * // Same as `createSeries({ data: [1, 2, 3, 4, 5] })`
 * const ser = series([1, 2, 3, 4, 5])
 * ```
 */
export const series = <
  TDataSource extends unknown[],
  TLabels extends readonly string[] = T.Readonly<
    T.Repeat<string, TDataSource['length']>
  >,
>(
  data: [...TDataSource],
) =>
  createSeries<TDataSource[number], TLabels>({
    data,
  });

type Max<
  TNumbers extends readonly number[],
  TAcc extends number = 0,
> = TNumbers extends [infer THead, ...infer TTail]
  ? THead extends number
    ? TTail extends readonly number[]
      ? N.Greater<THead, TAcc> extends 1
        ? Max<TTail, THead>
        : Max<TTail, TAcc>
      : never
    : never
  : TAcc;
/**
 * Creates a sparse series from an object of values, or an array of values and undefined
 * indicating the missing values. It is just another version of {@link
createSparseSeries}
 * with less options. If you don't care about labels, it is a cleaner way to create
a series.
 *
 * If you want to create a normal one, see {@link series}.
 * @param data The object of values, or an array of values and undefined.
```

```
 * @returns A sparse series.
 *
 * @see {@link createSparseSeries}
 *
 * @example
 * ```typescript
 * const ser1 = sparseSeries([42, undefined, 43]); // Series([42, <empty>, 43])
 *
 * // Note that the object keys are numbers (index), not strings (labels).
 * const ser1 = sparseSeries({ 1: 42 , 4: 43 }); // Series([<empty>, 42, <empty>, <empty>,
43])
 * ```
 */
export function sparseSeries<
  TDataSource extends unknown[],
  TLabels extends readonly string[] = T.Readonly<
    T.Repeat<string, TDataSource['length']>
  >,
>(
  data: [...TDataSource],
): SparseSeries<Exclude<TDataSource[number], undefined>, TLabels>;
export function sparseSeries<
  TDataSource extends Record<number, unknown>,
  TLabels extends readonly string[] = T.Readonly<
    T.Repeat<string, N.Add<Max<U.ListOf<keyof TDataSource>>, 1>>
  >,
>(
  data: TDataSource,
): TDataSource extends Record<number, infer TData>
  ? SparseSeries<TData, TLabels>
  : never;
export function sparseSeries<TData>(
  data: Record<number, TData> | Array<TData | undefined>,
): SparseSeries<TData> {
  if (Array.isArray(data)) {
    const result = createSparseSeries<TData>({ length: data.length });
    for (let i = 0; i < data.length; i++)
      if (data[i] !== undefined) result.$data[i] = data[i]!;
    return result;
  }

  const result = createSparseSeries<TData>({
    length: Object.keys(data)
      .map(Number)
      .reduce((a, b) => Math.max(a, b), 0),
```

```
  });
  for (const [key, val] of Object.entries(data))
    result.$data[key as unknown as number] = val;
  return result;
}
```

```
// series.test.ts
import { createSeries, createSparseSeries } from '../src/utils/series';

describe('Series', () => {
  const ser = createSeries({
    data: [1, 2, 3, 4, 5],
    labels: ['a', 'b', 'c', 'd', 'e'],
  });

  it('should be able to get a value by index', () => {
    expect(ser.at['d']).toBe(4);
    expect(ser.iat[3]).toBe(4);
  });

  it('should be able to set a value by index', () => {
    ser.at['d'] = 42;
    expect(ser.at['d']).toBe(42);
    ser.iat[3] = 4;
    expect(ser.iat[3]).toBe(4);
    expect(ser.at['d']).toBe(4);
  });

  it('should be able to transform a series', () => {
    const ser2 = ser.transform((val) => (val + 1).toString());
    expect(ser2.at['d']).toEqual('5');
  });

  it('should be able to accumulate a series', () => {
    const sum1 = ser.accumulate((acc, val) => acc - val, 0);
    expect(sum1).toBe(-15);
    const sum2 = ser.accumulate((acc, val) => acc - val);
    expect(sum2).toBe(-13);
    const sum3 = ser.accumulateRight((acc, val) => acc - val, 0);
    expect(sum3).toBe(-15);
    const sum4 = ser.accumulateRight((acc, val) => acc - val);
    expect(sum4).toBe(-5);
  });
});
```

```
describe('Sparse Series', () => {
  const labels = ['a', 'b', 'c', 'd', 'e'] as const;
  const ser = createSparseSeries<number>({ length: 5 }).withLabels(labels);

  it('should be able to set a value by index', () => {
    ser.at['e'] = 42;
    expect(ser.at['e']).toBe(42);
    ser.at['d'] = 43;
    expect(ser.at['d']).toBe(43);
    ser.iat[3] = 4;
    expect(ser.iat[3]).toBe(4);
    expect(ser.at['d']).toBe(4);
  });

  it('should be able to transform a series', () => {
    const ser2 = ser.transform((val) => (val + 1).toString());
    expect(ser2.at['d']).toEqual('5');
  });

  it('should be able to accumulate a series', () => {
    const sum1 = ser.accumulate((acc, val) => acc - val, 0);
    expect(sum1).toBe(-46);
    const sum2 = ser.accumulate((acc, val) => acc - val);
    expect(sum2).toBe(-38);
    const sum3 = ser.accumulateRight((acc, val) => acc - val, 0);
    expect(sum3).toBe(-46);
    const sum4 = ser.accumulateRight((acc, val) => acc - val);
    expect(sum4).toBe(38);
  });
});
```

```
// dataframe.ts
import fs from 'node:fs/promises';

import * as R from 'ramda';

import { createSeries } from './series.js';

import type { Series } from './series';
import type { Tuple as T, Union as U } from 'ts-toolbelt';

/**
 * The options for {@link DataFrame#toString}.
 *
 * @see {@link DataFrame#toString}
 */
interface DataFrameToStringOptions {
  /**
   * The maximum number of rows and columns to show.
   *
   * If it is a number, it will be used as the maximum number of rows (not columns).
   */
  limit?:
    | number
    | {
        /**
         * The maximum number of rows to show.
         * @default 10
         */
        row?: number;
        /**
         * The maximum number of columns to show.
         * @default 6
         */
        column?: number;
      };
  /**
   * The width of the header (i.e. the row names at the first column).
   * @default 20
   */
  headerWidth?: number;
  /**
   * The width of each column (except the first column).
   * @default 10
   */
  columnWidth?: number;
}

/**
 * A DataFrame is a 2-dimensional data structure that can store data of
 * different types (including characters, integers, floating point values,
 * categorical data and more) with labeled columns and (optionally labeled) rows.
 *
 * The data is stored by columns, and each column is a {@link Series},
 * so you should access the data by column if possible to avoid performance issues.
 * @template TData The type of the data stored in the DataFrame.
 * @template TColumnNames The type of the column names, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `col`
 * to `Series<TData, TRowNames, TType> | undefined`.
 * @template TRowNames The type of the row names, must be a tuple of strings.
 * Exact type is used if possible to ensure typesafety of accessing data,
 * otherwise fallback to `string[]`. When the type is `string[]`,
 * typesafety would still be ensured by Changing the return type of `row`
 * to `Series<TData, TColumnNames, TType> | undefined`.
 * @template TType The type of the DataFrame, can be either `'normal'` or `'sparse'`.
 *
 * @see {@link createDataFrame}
 */
export interface DataFrame<
  TData,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
> {
  /**
   * The type of the DataFrame, can be either `'normal'` or `'sparse'`.
   */
  readonly type: TType;
  /**
   * The shape of the DataFrame. The first element is the number of rows,
   * and the second element is the number of columns.
   *
   * If `TRowNames` is not the fallback type `string[]`,
   * then the type of the first element (i.e. row length) would be
   * the exact length of row names (literal number).
   * Otherwise, it would be just `number`.
   * The same applies to the second element (i.e. column length).
   *
   * It is a O(1) operation, whether the DataFrame is sparse or not.
   */
  readonly shape: [TRowNames['length'], TColumnNames['length']];
  /**
   * The names of the columns.
   */
  readonly columnNames: TColumnNames;
  /**
   * The names of the rows.
   */
  readonly rowNames: TRowNames;
  /**
   * The internal data of the DataFrame, stored as a list of Series.
   *
   * It is not recommended to access this property directly,
   * unless you know what you are doing.
   */
  readonly $data: Series<TData, TRowNames, TType>[];

  /**
   * Set all values of a column.
   *
   * It uses {@link Series#$setAll} internally, so the API is almost the same.
   * You can learn more about how to use it in {@link Series#$setAll}.
   * @param columnName The name of the column.
   * @param values Either an array of values, or a series. In case of sparse series,
   * the array can contain `undefined` to indicate missing values.
   * @throws If the length of `values` is not equal to the number of rows.
   *
   * @see {@link Series#$setAll}
   */
  $setColumn(
    columnName: TColumnNames[number],
    values:
      | (TType extends 'normal' ? TData[] : Array<TData | undefined>)
      | Series<TData, TRowNames, TType>,
  ): void;

  /**
   * Get or set a column at the given column name (not index).
   *
   * **Typesafety is ensured** when `TColumnNames` is the fallback type `string[]`,
   * i.e. the type of the return value is `Series<TData, TRowNames, TType> | undefined`.
   * While in exact type, the type of the return value is `Series<TData, TRowNames,
TType>`.
   *
   * @example
   * ```typescript
   * const df = createDataFrame({
   *   data: [[1, 2, 3], [4, 5, 6]],
   *   columnNames: ['A', 'B', 'C'],
   * });
   * df.col['B']; // Series([2, 5])
   * df.col['B'] = series([42, 43]); // `df` is now [[1, 42, 3], [4, 43, 6]]
   * df.col['B']; // Series([42, 43])
   * ```
   */
  col: {
```

```
    [ColumnName in TColumnNames[number]]: string[] extends TColumnNames
      ? Series<TData, TRowNames, TType> | undefined
      : Series<TData, TRowNames, TType>;
  };
  /**
   * Get or set a column at the given integer position of the column,
   * i.e. column index (not name).
   *
   * Typesafety is **not** ensured when `TColumnNames` is the fallback type `string[]`,
   * so the type of the return value is always `Series<TData, TRowNames, TType>`.
   * You have to ensure the index is valid by yourself.
   *
   * @example
   * ```typescript
   * const df = createDataFrame({
   *   data: [[1, 2, 3], [4, 5, 6]],
   *   columnNames: ['A', 'B', 'C'],
   * });
   * df.icol[1]; // Series([2, 5])
   * df.icol[1] = series([42, 43]); // `df` is now [[1, 42, 3], [4, 43, 6]]
   * df.icol[1]; // Series([42, 43])
   * ```
   */
  icol: {
    [key: number]: Series<TData, TRowNames, TType>;
  };

  /**
   * Get or set a row at the given row name (not index).
   *
   * **Typesafety is ensured** when `TRowNames` is the fallback type `string[]`,
   * i.e. the type of the return value is `Series<TData, TColumnNames, TType> | undefined`.
   * While in exact type, the type of the return value is `Series<TData, TColumnNames, TType>`.
   *
   * Accessing data by row should **not** be used as your main way to access data,
   * as DataFrame is column-oriented, and accessing data by row is much slower than
   * accessing data by column. Consider using {@link DataFrame#col} instead if possible,
   * or use {@link DataFrame#transpose} to transpose the DataFrame first and then
   * access data by column for better performance.
   *
   * @example
   * ```typescript
   * const df = createDataFrame({
```

```
 *   data: [[1, 2, 3], [4, 5, 6]],
 *   columnNames: ['A', 'B', 'C'],
 *   rowNames: ['r1', 'r2'],
 * });
 * df.row['r2']; // Series([4, 5, 6])
 * df.row['r2'] = series([42, 43, 44]); // `df` is now [[1, 2, 3], [42, 43, 44]]
 * df.row['r2']; // Series([42, 43, 44])
 * ```
 */
row: {
  [RowName in TRowNames[number]]: string[] extends TRowNames
    ? Series<TData, TColumnNames, TType> | undefined
    : Series<TData, TColumnNames, TType>;
};
/**
 * Get or set a row at the given integer position of the row,
 * i.e. row index (not name).
 *
 * Typesafety is **not** ensured when `TRowNames` is the fallback type `string[]`,
 * so the type of the return value is always `Series<TData, TColumnNames, TType>`.
 * You have to ensure the index is valid by yourself.
 *
 * Accessing data by row should **not** be used as your main way to access data,
 * as DataFrame is column-oriented, and accessing data by row is much slower than
 * accessing data by column. Consider using {@link DataFrame#col} instead if possible,
 * or use {@link DataFrame#transpose} to transpose the DataFrame first and then
 * access data by column for better performance.
 *
 * @example
 * ```typescript
 * const df = createDataFrame({
 *   data: [[1, 2, 3], [4, 5, 6]],
 *   columnNames: ['A', 'B', 'C'],
 * });
 * df.irow[1]; // Series([4, 5, 6])
 * df.irow[1] = series([42, 43, 44]); // `df` is now [[1, 2, 3], [42, 43, 44]]
 * df.irow[1]; // Series([42, 43, 44])
 * ```
 */
irow: { [key: number]: Series<TData, TColumnNames, TType> };

/**
 * Transpose the DataFrame, i.e. swap the row and column.
 * @returns A new DataFrame with row and column swapped.
 *
```

```typescript
 * @example
 * ```typescript
 * const df = createDataFrame({
 *   data: [[1, 2, 3], [4, 5, 6]],
 *   columnNames: ['A', 'B', 'C'],
 *   rowNames: ['r1', 'r2'],
 * });
 * // `newDf` is now [[1, 4], [2, 5], [3, 6]]
 * // with column names ['r1', 'r2'] and row names ['A', 'B', 'C']
 * const newDf = df.transpose();
 * ```
 */
transpose(): DataFrame<TData, TRowNames, TColumnNames, TType>;

/**
 * Clone the DataFrame.
 * @returns A new DataFrame with the same data.
 */
clone(): DataFrame<TData, TColumnNames, TRowNames, TType>;

/**
 * Create a new DataFrame with the given column names.
 * @param columnNames The new column names.
 * @returns A new DataFrame with the given column names.
 */
withColumnNames<const NTColumnNames extends readonly string[]>(
  columnNames: NTColumnNames,
): NTColumnNames['length'] extends TColumnNames['length']
  ? DataFrame<TData, NTColumnNames, TRowNames, TType>
  : never;
/**
 * Create a new DataFrame with the given row names.
 * @param rowNames The new row names.
 * @returns A new DataFrame with the given row names.
 */
withRowNames<const NTRowNames extends readonly string[]>(
  rowNames: NTRowNames,
): NTRowNames['length'] extends TRowNames['length']
  ? DataFrame<TData, TColumnNames, NTRowNames, TType>
  : never;

[Symbol.toStringTag]: 'DataFrame';
/**
 * Convert the DataFrame to a human-readable string.
 * @param options Options for converting the DataFrame to string.
```

```
   *
   * @see {@link DataFrameToStringOptions}
   */
  toString(options?: DataFrameToStringOptions): string;
  /**
   * Print the DataFrame to the console in a human-readable format.
   *
   * It uses {@link DataFrame#toString} internally.
   * @param options Options for converting the DataFrame to string.
   *
   * @see {@link DataFrameToStringOptions}
   */
  show(options?: DataFrameToStringOptions): void;
  /**
   * Print the first few rows of the DataFrame to the console in a human-readable format.
   *
   * It uses {@link DataFrame#show} internally.
   * @param limit The number of rows to print. Defaults to 5.
   */
  showHead(limit?: number): void;

  /**
   * Save the DataFrame to a file.
   * @param pathname Path to the file to save.
   */
  saveToFile(pathname: string): Promise<void>;
}
/**
 * The sparse version of `DataFrame`.
 * It is just the same to `DataFrame<TData, TColumnNames, TRowNames, 'sparse'>`.
 */
export type SparseDataFrame<
  TData,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
> = DataFrame<TData, TColumnNames, TRowNames, 'sparse'>;

interface CreateDataFrameCommonOptions<
  TColumnNames extends readonly string[],
  TRowNames extends readonly string[],
> {
  columnNames: TColumnNames;
  rowNames?: TRowNames;
}
interface CreateDataFrameMatrixNormalOptions<TData> {
  data: TData[][];
}
interface CreateDataFrameMatrixSparseOptions {
  shape: [number, number];
}
type CreateDataFrameMatrixOptions<TData, TType extends 'normal' | 'sparse'> = {
  type?: TType;
} & (TType extends 'normal'
  ? CreateDataFrameMatrixNormalOptions<TData>
  : CreateDataFrameMatrixSparseOptions);
/**
 * The options for {@link createDataFrame}.
 *
 * @see {@link createDataFrame}
 */
export type CreateDataFrameOptions<
  TData,
  TColumnNames extends readonly string[],
  TRowNames extends readonly string[],
  TType extends 'normal' | 'sparse',
> = CreateDataFrameCommonOptions<TColumnNames, TRowNames> &
  CreateDataFrameMatrixOptions<TData, TType>;
/**
 * The options for {@link createSparseDataFrame}.
 *
 * @see {@link createSparseDataFrame}
 */
export type CreateSparseDataFrameOptions<
  TColumnNames extends readonly string[],
  TRowNames extends readonly string[],
> = CreateDataFrameCommonOptions<TColumnNames, TRowNames> &
  CreateDataFrameMatrixSparseOptions;

/**
 * Create a DataFrame (normal or sparse) from the given options.
 * @param options The options to create the DataFrame.
 * @returns A DataFrame.
 *
 * @example
 * ```typescript
 * // Create a normal DataFrame (columns are stored in arrays)
 * const df = createDataFrame({
 *   data: [[1, 2, 3], [4, 5, 6]],
 *   columnNames: ['A', 'B', 'C'],
 * });
```

```
 *
 * // Create a sparse DateFrame (columns are stored in an objects)
 * // `data` cannot be provided in this case, instead `shape` must be provided
 * // to specify the row length and column length of the DataFrame
 * // Note that in this case the data type (i.e. `TData`) cannot be inferred
 * // by TypeScript, so you have to specify it explicitly
 * // Also, the type of column names (i.e. `TColumnNames`)
 * // and row names (i.e. `TRowNames`) should be provided by your own
 * // if you want to keep them exact
 * const columnNames = ['A', 'B', 'C'] as const;
 * const rowNames = ['r1', 'r2'] as const;
 * const sparseDf = createDataFrame<number, typeof columnNames, typeof rowNames>({
 *   type: 'sparse',
 *   shape: [2, 3],
 *   columnNames,
 *   rowNames,
 * });
 * ```
 */
export const createDataFrame = <
  TData,
  const TColumnNames extends readonly string[],
  const TRowNames extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
>(
  options: CreateDataFrameOptions<TData, TColumnNames, TRowNames, TType>,
): DataFrame<TData, TColumnNames, TRowNames, TType> => {
  const { columnNames, rowNames } = options;

  if (columnNames && R.uniq(columnNames).length !== columnNames.length)
    throw new Error('Column names must be unique');
  if (rowNames && R.uniq(rowNames).length !== rowNames.length)
    throw new Error('Row names must be unique');

  const rowLength =
    options.type === 'sparse'
      ? (options as CreateDataFrameMatrixSparseOptions).shape[0]
      : (options as CreateDataFrameMatrixNormalOptions<TData>).data.length;
  const columnLength =
    options.type === 'sparse'
      ? (options as CreateDataFrameMatrixSparseOptions).shape[1]
      : (options as CreateDataFrameMatrixNormalOptions<TData>).data[0]
          ?.length ?? 0;

  if (columnNames && columnNames.length !== columnLength)
```

```
    throw new Error('Column length must match matrix column length');
  if (rowNames && rowNames.length !== rowLength)
    throw new Error('row length must match matrix row length');

  const _columnNames = [...columnNames] as TColumnNames;
  Object.freeze(_columnNames);
  const _rowNames = (
    rowNames !== undefined ? [...rowNames] : R.range(0, rowLength).map(String)
  ) as TRowNames;
  Object.freeze(_rowNames);

  const _columnName2columnIndex = R.fromPairs(
    _columnNames.map((columnName, i) => [columnName, i]),
  ) as Record<TColumnNames[number], number>;
  const _rowName2rowIndex = R.fromPairs(
    _rowNames.map((rowName, i) => [rowName, i]),
  ) as Record<TRowNames[number], number>;

  const _data = (
    options.type === 'sparse'
      ? Array.from({ length: columnLength }, () =>
          createSeries<TData, TRowNames, 'sparse'>({
            type: options.type as 'sparse',
            length: rowLength,
            labels: _rowNames,
            $checkLabels: false,
            $copyLabels: false,
            $label2indexMap: _rowName2rowIndex,
          }),
        )
      : (options as CreateDataFrameMatrixNormalOptions<TData>).data.reduce(
          (acc, row, rowIndex) => {
            row.forEach((val, columnIndex) => {
              acc[columnIndex].$data[rowIndex] = val;
            });
            return acc;
          },
          Array.from({ length: columnLength }, () =>
            createSeries({
              type: options.type as 'normal' | undefined,
              data: Array.from(
                { length: rowLength },
                () => undefined as unknown as TData,
              ),
              labels: _rowNames,
              $checkLabels: false,
              $copyLabels: false,
              $label2indexMap: _rowName2rowIndex,
            }),
          ),
        )
  ) as Series<TData, TRowNames, TType>[];

  const _saveNormal = async (pathname: string) => {
    let content = 'normal,' + _rowNames.join(',') + '\n';
    for (const [columnIndex, series] of _data.entries()) {
      content +=
        _columnNames[columnIndex] +
        ',' +
        (series as Series<TData, TRowNames, 'normal'>).$data.join(',') +
        '\n';
    }
    content = content.slice(0, -1);
    await fs.writeFile(pathname, content);
  };

  const _saveSparse = async (pathname: string) => {
    let content = 'sparse,' + _rowNames.join(',') + '\n';
    for (const [columnIndex, series] of _data.entries()) {
      content +=
        _columnNames[columnIndex] + ',' + JSON.stringify(series.$data) + '\n';
    }
    content = content.slice(0, -1);
    await fs.writeFile(pathname, content);
  };

  const result: DataFrame<TData, TColumnNames, TRowNames, TType> = {
    get type() {
      return (options.type ?? 'normal') as TType;
    },
    get shape() {
      return [rowLength, columnLength] as [number, number];
    },
    get columnNames() {
      return _columnNames;
    },
    get rowNames() {
      return _rowNames;
    },
    get $data() {
      return _data;
    },

    $setColumn(columnName, values) {
      if (values.length !== rowLength)
        throw new Error('Value length must match row length');
      const ser = _data[_columnName2columnIndex[columnName]];
      ser.$setAll(values);
    },

    col: new Proxy(
      {},
      {
        get(_, columnName: TColumnNames[number]) {
          return _data[_columnName2columnIndex[columnName]];
        },
        set(_, columnName: TColumnNames[number], val) {
          _data[_columnName2columnIndex[columnName]].$setAll(val.toArray());
          return true;
        },
      },
    ) as DataFrame<TData, TColumnNames, TRowNames, TType>['col'],
    icol: new Proxy(
      {},
      {
        get(_, columnIndex) {
          return _data[columnIndex as unknown as number];
        },
        set(_, columnIndex, val) {
          _data[columnIndex as unknown as number].$setAll(val.toArray());
          return true;
        },
      },
    ),

    row: new Proxy(
      {},
      {
        get(_, rowName: TRowNames[number]) {
          if (options.type === 'sparse') {
            const result = createSeries<TData, TColumnNames, 'sparse'>({
              type: 'sparse',
              length: columnLength,
              labels: _columnNames,
              $checkLabels: false,
              $copyLabels: false,
              $label2indexMap: _columnName2columnIndex,
            });
            for (const [columnIndex, series] of _data.entries()) {
              const val = series.$data[_rowName2rowIndex[rowName]];
              if (val !== undefined) result.$data[columnIndex] = val;
            }
            return result;
          }

          const newData = _data.map(
            (ser) => ser.$data[_rowName2rowIndex[rowName]],
          );
          return createSeries<TData, TColumnNames, 'normal'>({
            type: 'normal',
            data: newData,
            labels: _columnNames,
            $checkLabels: false,
            $copyLabels: false,
            $label2indexMap: _columnName2columnIndex,
          });
        },

        set(_, rowName: TRowNames[number], val) {
          if (options.type === 'sparse') {
            for (const [columnIndex, ser] of _data.entries()) {
              const value = val.$data[columnIndex];
              if (value !== undefined)
                ser.$data[_rowName2rowIndex[rowName]] = value;
            }
            return true;
          }

          for (const [columnIndex, ser] of _data.entries()) {
            ser.$data[_rowName2rowIndex[rowName]] = val.$data[columnIndex];
          }
          return true;
        },
      },
    ) as DataFrame<TData, TColumnNames, TRowNames, TType>['row'],
    irow: new Proxy(
      {},
      {
        get(_, rowIndex) {
          if (options.type === 'sparse') {
```

```
        const result = createSeries<TData, TColumnNames, 'sparse'>({
          type: 'sparse',
          length: columnLength,
          labels: _columnNames,
          $checkLabels: false,
          $copyLabels: false,
          $label2indexMap: _columnName2columnIndex,
        });
        for (const [columnIndex, series] of _data.entries()) {
          const val = series.$data[rowIndex as unknown as number];
          if (val !== undefined) result.$data[columnIndex] = val;
        }
        return result;
      }

      const newData = _data.map(
        (ser) => ser.$data[rowIndex as unknown as number],
      );
      return createSeries<TData, TColumnNames, 'normal'>({
        type: 'normal',
        data: newData,
        labels: _columnNames,
        $checkLabels: false,
        $copyLabels: false,
        $label2indexMap: _columnName2columnIndex,
      });
    },

    set(_, rowIndex, val) {
      if (options.type === 'sparse') {
        for (const [columnIndex, ser] of _data.entries()) {
          const value = val.$data[columnIndex];
          if (value !== undefined)
            ser.$data[rowIndex as unknown as number] = value;
        }
        return true;
      }

      for (const [columnIndex, ser] of _data.entries()) {
        ser.$data[rowIndex as unknown as number] = val.$data[columnIndex];
      }
      return true;
    },
  },
) as { [key: number]: Series<TData, TColumnNames, TType> },
transpose() {
  if (options.type === 'sparse') {
    const result = createDataFrame<
      TData,
      TRowNames,
      TColumnNames,
      'sparse'
    >({
      ...options,
      type: options.type as 'sparse',
      shape: [columnLength, rowLength],
      columnNames: _rowNames as unknown as TRowNames,
      rowNames: _columnNames as unknown as TColumnNames,
    });
    for (const [columnIndex, series] of _data.entries()) {
      for (const [rowIndex, value] of Object.entries(series.$data)) {
        result.$data[Number(rowIndex)].$data[columnIndex] = value;
      }
    }
    return result as DataFrame<TData, TRowNames, TColumnNames, TType>;
  }

  return createDataFrame<TData, TRowNames, TColumnNames, 'normal'>({
    ...options,
    type: options.type as 'normal',
    data: _data.map((ser) => ser.$data) as TData[][],
    columnNames: _rowNames as unknown as TRowNames,
    rowNames: _columnNames as unknown as TColumnNames,
  }) as DataFrame<TData, TRowNames, TColumnNames, TType>;
},

clone() {
  if (options.type === 'sparse') {
    const result = createDataFrame(options);
    for (const [columnIndex, series] of _data.entries()) {
      for (const [rowIndex, value] of Object.entries(series.$data)) {
        result.$data[rowIndex as unknown as number].$data[columnIndex] =
          value;
      }
    }
    return result;
  }

  return createDataFrame<TData, TColumnNames, TRowNames, TType>({
    ...options,
    data: _data.map((ser) => ser.$data),
  });
},

withColumnNames(columnNames) {
  if (options.type === 'sparse') {
    const result = createDataFrame({
      ...options,
      columnNames,
    });
    for (const [columnIndex, series] of _data.entries()) {
      for (const [rowIndex, val] of Object.entries(series.$data)) {
        result.$data[rowIndex as unknown as number].$data[columnIndex] =
          val;
      }
    }
    // eslint-disable-next-line @typescript-eslint/no-explicit-any
    return result as any;
  }

  return createDataFrame({
    ...options,
    columnNames,
    data: R.range(0, rowLength).map((rowIndex) =>
      _data.map((ser) => ser.$data[rowIndex]),
    ),
    // eslint-disable-next-line @typescript-eslint/no-explicit-any
  }) as any;
},
withRowNames(rowNames) {
  if (options.type === 'sparse') {
    const result = createDataFrame({
      ...options,
      rowNames,
    });
    for (const [columnIndex, series] of _data.entries()) {
      for (const [rowIndex, val] of Object.entries(series.$data)) {
        result.$data[rowIndex as unknown as number].$data[columnIndex] =
          val;
      }
    }
    // eslint-disable-next-line @typescript-eslint/no-explicit-any
    return result as any;
  }
  return createDataFrame({
    ...options,
    rowNames,
    data: R.range(0, rowLength).map((rowIndex) =>
      _data.map((ser) => ser.$data[rowIndex]),
    ),
    // eslint-disable-next-line @typescript-eslint/no-explicit-any
  }) as any;
},

[Symbol.toStringTag]: 'DataFrame',
toString({
  columnWidth = 10,
  headerWidth = 20,
  limit: inputLimit,
}: DataFrameToStringOptions = {}) {
  const limit = {
    column: 6,
    row: 10,
  };
  if (inputLimit !== undefined) {
    if (typeof inputLimit === 'number') {
      limit.row = inputLimit;
    } else {
      if ('row' in inputLimit && inputLimit.row !== undefined)
        limit.row = inputLimit.row;
      if ('column' in inputLimit && inputLimit.column !== undefined)
        limit.column = inputLimit.column;
    }
  }

  let result = '';

  result += ' '.repeat(headerWidth);
  result += _columnNames
    .slice(0, limit.column)
    .map((col) => col.padStart(columnWidth))
    .join('');
  result += '\n';

  for (let rowIndex = 0; rowIndex < rowLength; rowIndex++) {
    if (rowIndex >= limit.row) break;

    const row = _data.map((ser) => ser.$data[rowIndex]);
    result += _rowNames[rowIndex].padEnd(headerWidth);
    result += (
      options.type === 'sparse'
        ? R.range(0, columnLength).map(
            (columnIndex) => row[columnIndex] ?? '<empty>',
          )
        : row
    )
      .slice(0, limit.column)
      .map((val) =>
        (typeof val === 'number' ? val.toFixed(4) : String(val)).padStart(
          columnWidth,
        ),
      )
      .join('');

    if (columnLength > limit.column) {
      result += `    ... ${columnLength - limit.column} more columns`;
    }

    result += '\n';
  }

  if (rowLength > limit.row) {
    result += `... ${rowLength - limit.row} more rows`;
  } else {
    result = result.slice(0, -1);
  }

  return result;
},
show(options) {
  console.log(result.toString(options));
},
showHead(limit = 5) {
  result.show({ limit: { row: limit } });
},

async saveToFile(pathname) {
  if (options.type === 'sparse') await _saveSparse(pathname);
  else await _saveNormal(pathname);
},
};
```

```
  return result;
};
/**
 * Creates a sparse DataFrame. It is just the same when passing
 * `type: 'sparse'` to `createDataFrame`.
 * @param options
 * @returns A sparse DataFrame.
 *
 * @see {@link createDataFrame}
 */
export const createSparseDataFrame = <
  TData,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
>(
  options: CreateSparseDataFrameOptions<TColumnNames, TRowNames>,
): SparseDataFrame<TData, TColumnNames, TRowNames> =>
  createDataFrame<TData, TColumnNames, TRowNames, 'sparse'>({
    ...options,
    type: 'sparse',
  });

/**
 * Load a DataFrame from a file.
 *
 * It is normally used to load a DataFrame saved by {@link DataFrame#saveToFile}
 * @param pathname The path to the file.
 * @returns The loaded DataFrame.
 *
 * @see {@link DataFrame#saveToFile}
 */
export const loadDataFrame = async <
  TData extends number | string,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
  TType extends 'normal' | 'sparse' = 'normal',
>(
  pathname: string,
): Promise<DataFrame<TData, TColumnNames, TRowNames, TType>> => {
  const parseNormal = (
    lines: string[],
  ): DataFrame<TData, TColumnNames, TRowNames, 'normal'> => {
    const rowNames = lines[0].split(',').slice(1) as unknown as TRowNames;
    const columnNames = lines
      .slice(1)
    .map((line) => line.split(',', 2)[0]) as unknown as TColumnNames;

  const matrix = lines.slice(1).map((line) =>
    line
      .split(',')
      .slice(1)
      .map((val) => (/^\d+(\.\d+)?$/.test(val) ? Number(val) : val)),
  ) as unknown as TData[][];

  return createDataFrame({
    data: matrix.reduce(
      (acc, series) => {
        series.forEach((val, i) => acc[i].push(val));
        return acc;
      },
      Array.from({ length: rowNames.length }, () => [] as TData[]),
    ),
    columnNames,
    rowNames,
  });
};

const parseSparse = (
  lines: string[],
): DataFrame<TData, TColumnNames, TRowNames, 'sparse'> => {
  const rowNames = lines[0].split(',').slice(1) as unknown as TRowNames;
  const columnNames = lines
    .slice(1)
    .map((line) => line.split(',', 2)[0]) as unknown as TColumnNames;

  const result = createDataFrame<TData, TColumnNames, TRowNames, 'sparse'>({
    type: 'sparse',
    shape: [rowNames.length, columnNames.length],
    columnNames,
    rowNames,
  });

  for (let columnIndex = 0; columnIndex < columnNames.length; columnIndex++) {
    const line = lines[columnIndex + 1];
    const json = line.slice(line.split(',', 2)[0].length + 1);

    for (const [rowIndex, val] of Object.entries(JSON.parse(json))) {
      result.$data[columnIndex].$data[rowIndex as unknown as number] =
        val as TData;
    }
    }

    return result;
  };

  const content = await fs.readFile(pathname, 'utf8');
  const lines = content.split('\n').filter((line) => line.length > 0);

  const type = lines[0].split(',', 2)[0].trim();
  if (type === 'normal')
    return parseNormal(lines) as DataFrame<
      TData,
      TColumnNames,
      TRowNames,
      TType
    >;
  return parseSparse(lines) as DataFrame<TData, TColumnNames, TRowNames, TType>;
};
/**
 * Load a sparse version of DataFrame from a file.
 * It is just the same to `loadDataFrame`, but with default typing
 * of `type` to `sparse`.
 * @param pathname The path to the file.
 * @returns The loaded DataFrame.
 *
 * @see {@link loadDataFrame}
 */
export const loadSparseDataFrame = <
  TData extends number | string,
  TColumnNames extends readonly string[] = string[],
  TRowNames extends readonly string[] = string[],
>(
  pathname: string,
): Promise<SparseDataFrame<TData, TColumnNames, TRowNames>> =>
  loadDataFrame<TData, TColumnNames, TRowNames, 'sparse'>(pathname);

/**
 * Creates a normal DataFrame from an object with keys as column names and values as
 * arrays. It is just another version of {@link createDataFrame} with less options.
 * If you don't care about row names, it is a cleaner way to create a DataFrame.
 *
 * If you want to create a sparse one, see {@link sparseDataFrame}.
 * @param data The object with keys as column names and values as arrays.
 * @returns A normal DataFrame.
 *
 * @see {@link createDataFrame}
 *
 * @example
 * ```typescript
 * // Same as `createDataFrame({ data: [[1, 2, 3], [4, 5, 6]], columnNames: ['A', 'B', 'C'] })`
 * const df = dataFrame({
 *   A: [1, 4],
 *   B: [2, 5],
 *   C: [3, 6],
 * });
 * ```
 */
export const dataFrame = <
  TDataSource extends Record<string, unknown[]>,
  TColumnNames extends readonly string[] = T.Readonly<
    U.ListOf<keyof TDataSource>
  >,
  TRowNames extends readonly string[] = string[],
>(
  data: TDataSource,
) => {
  const columnNames = Object.keys(data) as unknown as TColumnNames;
  const rowLength =
    columnNames[0] !== undefined ? data[columnNames[0]].length : 0;

  return createDataFrame<
    TDataSource[keyof TDataSource][number],
    TColumnNames,
    TRowNames
  >({
    data: R.range(0, rowLength).map((i) =>
      columnNames.map((columnName) => data[columnName][i]),
    ) as unknown as TDataSource[keyof TDataSource][][],
    columnNames,
  });
};
/**
 * Creates a sparse DateFrame from an object with keys as column names and values as
 * values of the series or an array of values and undefined indicating the missing values.
 * It is just another version of {@link createSparseDataFrame} with less options.
 * If you don't care about row names, it is a cleaner way to create a DataFrame.
 *
 * If you want to create a normal one, see {@link dataFrame}.
```

```
 * @param data The object with keys as column names and values as values of the series
 * or an array of values and undefined indicating the missing values.
 * @returns A sparse series.
 *
 * @see {@link createSparseDataFrame}
 *
 * @example
 * ```typescript
 * // Create a sparse DataFrame [[1, undefined], [undefined, 42], [3, 43]]
 * const df = sparseDataFrame({
 *   A: [1, undefined, 3],
 *   // Note that the object keys are numbers (index), not strings (labels).
 *   B: { 1: 42, 2: 43 },
 * });
 * ```
 */
export const sparseDataFrame = <
  TDataSource extends Record<string, unknown[] | Record<number, unknown>>,
  TColumnNames extends readonly string[] = T.Readonly<
    U.ListOf<keyof TDataSource>
  >,
  TRowNames extends readonly string[] = string[],
>(
  data: TDataSource,
) => {
  const columnNames = Object.keys(data) as unknown as TColumnNames;
  const rowLength = columnNames.reduce(
    (acc, columnName) =>
      Math.max(
        acc,
        Array.isArray(data[columnName])
          ? (data[columnName] as unknown[]).length
          : Object.keys(data[columnName] as Record<number, unknown>).reduce(
              (acc, key) => Math.max(acc, Number(key)),
              0,
            ),
      ),
    0,
  );

  const result = createSparseDataFrame<
    TDataSource[keyof TDataSource][number],
    TColumnNames,
    TRowNames
  >({
```

```
    shape: [rowLength, columnNames.length],
    columnNames,
  });

  for (let columnIndex = 0; columnIndex < columnNames.length; columnIndex++) {
    const columnName = columnNames[columnIndex];
    const columnData = data[columnName];
    if (Array.isArray(columnData)) {
      for (let rowIndex = 0; rowIndex < columnData.length; rowIndex++) {
        if (columnData[rowIndex] !== undefined)
          result.$data[columnIndex].$data[rowIndex] = columnData[
            rowIndex
          ] as TDataSource[keyof TDataSource][number];
      }
    } else {
      for (const [rowIndex, val] of Object.entries(columnData)) {
        result.$data[columnIndex].$data[rowIndex as unknown as number] =
          val as TDataSource[keyof TDataSource][number];
      }
    }
  }

  return result;
};
```

```
// dataframe.test.ts
import fs from 'node:fs/promises';

import {
  dataFrame,
  loadDataFrame,
  sparseDataFrame,
} from '../src/utils/dataframe';
import { series, sparseSeries } from '../src/utils/series';

describe('DataFrame', () => {
  let fileContent: string;

  beforeAll(() => {
    jest.spyOn(fs, 'writeFile').mockImplementation(async (_, content) => {
      fileContent = content as string;
    });
    jest.spyOn(fs, 'readFile').mockImplementation(async () => fileContent);
  });
```

```
afterAll(() => {
  jest.restoreAllMocks();
});

const df = dataFrame({
  c1: [1, 4],
  c2: [2, 5],
  c3: [3, 6],
}).withRowNames(['r1', 'r2']);

test('get', () => {
  expect(df.col['c1'].at['r1']).toBe(1);
  expect(df.col['c2'].at['r2']).toBe(5);
});

test('set', () => {
  df.col['c1'].at['r1'] = 10;
  expect(df.col['c1'].at['r1']).toBe(10);
  df.col['c1'].at['r1'] = 1;
  expect(df.col['c1'].at['r1']).toBe(1);
});

test('getColumn', () => {
  expect(df.col['c1'].toArray()).toEqual([1, 4]);
  expect(df.col['c2'].toArray()).toEqual([2, 5]);
});

test('setColumn', () => {
  df.col['c2'] = series([10, 20]);
  expect(df.col['c2'].toArray()).toEqual([10, 20]);
  df.col['c2'] = series([2, 5]);
  expect(df.col['c2'].toArray()).toEqual([2, 5]);
});

it('should be able to get a row by name', () => {
  expect(df.row['r1'].toArray()).toEqual([1, 2, 3]);
});

it('should be able to set a row by name', () => {
  df.row['r1'] = series([10, 20, 30]);
  expect(df.row['r1'].toArray()).toEqual([10, 20, 30]);
  df.row['r1'] = series([1, 2, 3]);
  expect(df.row['r1'].toArray()).toEqual([1, 2, 3]);
});
```

```
  it('should be able to get a row by index', () => {
    expect(df.irow[0].toArray()).toEqual([1, 2, 3]);
  });

  it('should be able to set a row by index', () => {
    df.irow[0] = series([10, 20, 30]);
    expect(df.irow[0].toArray()).toEqual([10, 20, 30]);
    df.irow[0] = series([1, 2, 3]);
    expect(df.irow[0].toArray()).toEqual([1, 2, 3]);
  });

  test('toString', () => {
    const str = df.toString();
    expect(str.split('\n').length).toBe(3);
  });

  test('save', async () => {
    await df.saveToFile('test.csv');
    expect(fs.writeFile).toBeCalled();
  });

  test('load', async () => {
    const df2: typeof df = await loadDataFrame('test.csv');
    expect(fs.readFile).toBeCalled();
    expect(df2.rowNames).toEqual(df.rowNames);
    expect(df2.columnNames).toEqual(df.columnNames);
    expect(df2.col['c1'].at['r1']).toEqual(df.col['c1'].at['r1']);
    expect(df2.col['c2'].at['r2']).toEqual(df.col['c2'].at['r2']);
  });
});

describe('Sparse DataFrame', () => {
  let fileContent: string;

  beforeAll(() => {
    jest.spyOn(fs, 'writeFile').mockImplementation(async (_, content) => {
      fileContent = content as string;
    });
    jest.spyOn(fs, 'readFile').mockImplementation(async () => fileContent);
  });

  afterAll(() => {
    jest.restoreAllMocks();
  });
```

```
const df = sparseDataFrame({
  c1: [undefined, undefined],
  c2: {},
  c3: {},
}).withRowNames(['r1', 'r2']);

it('should be able to set a value by column name and row name', () => {
  df.col['c3'].at['r2'] = 42;
  expect(df.col['c3'].at['r2']).toBe(42);
});

it('should be able to get a column by column name', () => {
  expect(df.col['c3'].toArray()).toEqual([undefined, 42]);
});

it('should be able to set a column by index', () => {
  df.col['c3'] = sparseSeries([undefined, 43]);
  expect(df.col['c3'].toArray()).toEqual([undefined, 43]);
  df.col['c3'] = sparseSeries([undefined, 42]);
  expect(df.col['c3'].toArray()).toEqual([undefined, 42]);
});

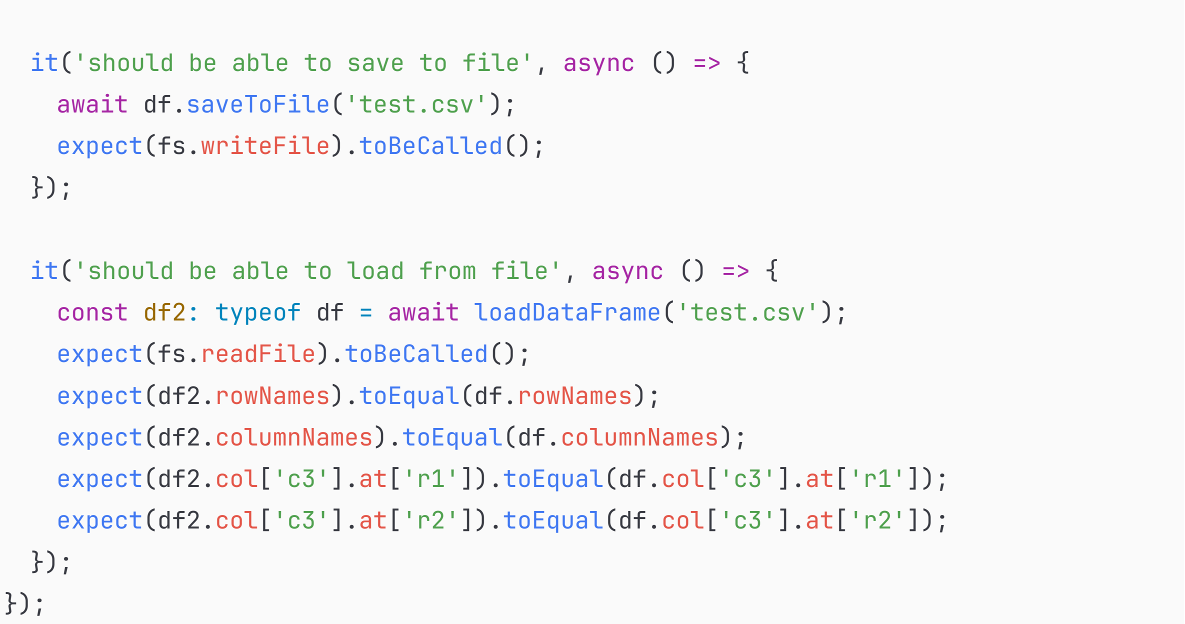
it('should be able to get a row by name', () => {
  expect(df.row['r2'].toArray()).toEqual([undefined, undefined, 42]);
});

it('should be able to set a row by name', () => {
  df.row['r1'] = sparseSeries([undefined, undefined, 30]);
  expect(df.row['r1'].toArray()).toEqual([undefined, undefined, 30]);
});

it('should be able to get a row by index', () => {
  expect(df.irow[0].toArray()).toEqual([undefined, undefined, 30]);
});

it('should be able to set a row by index', () => {
  df.irow[0] = sparseSeries([undefined, undefined, 40]);
  expect(df.irow[0].toArray()).toEqual([undefined, undefined, 40]);
});

it('should be able to convert to string', () => {
  const str = df.toString();
  expect(str.split('\n').length).toBe(3);
});
```



```
  it('should be able to save to file', async () => {
    await df.saveToFile('test.csv');
    expect(fs.writeFile).toBeCalled();
  });

  it('should be able to load from file', async () => {
    const df2: typeof df = await loadDataFrame('test.csv');
    expect(fs.readFile).toBeCalled();
    expect(df2.rowNames).toEqual(df.rowNames);
    expect(df2.columnNames).toEqual(df.columnNames);
    expect(df2.col['c3'].at['r1']).toEqual(df.col['c3'].at['r1']);
    expect(df2.col['c3'].at['r2']).toEqual(df.col['c3'].at['r2']);
  });
});
```

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。与 Pandas 类似，DataFrame 使用 Series 按列存储数据。同时，该 DataFrame 与 Series 支持对应的稀疏版本，即使用 JS 对象（可以理解为 Python 中的字典）存储数据，以降低内存占用。除此之外，该 DataFrame 与 Series 支持一些常见的方法、如克隆、转置、变换等。

2. 本次实验最大的难点在于该如何在有限的内存中处理所给的大量数据。这里使用的方法是通过一个稀疏矩阵存储中间向量，以降低内存占用。同时，稀疏矩阵在稀疏情况下的处理速度也要快速紧凑矩阵，因其只需要遍历存储的值，而不需要遍历所有值。

3. 另一个难点在于如何处理两个长度可能不等的向量的余弦相似度。在这里，由于使用的方式（Series）中向量已经保留了单词标签，这个问题比较自然地解决了，不需要过多考虑。

4. 除此之外，本次实验还存在一些潜在的性能问题。在对大量文档的查询中，任何不必要的开销都将对查询性能产生显著影响。事实上，DataFrame 的最初版本中包含了一些不必要的 O(n)操作，这显著降低了查询性能，一次查询需要约半小时才能完成。在优化了所有不必要的 O(n)操作后，一次查询只需要约 2-5 秒即可完成。若考虑将性能开销较大的计算转移到高性能语言如 C++或 Rust 中，并使用显卡介入部分计算，性能应当有很大的提升空间。

5. 通过本次实验，进一步实践了文档相似度的计算，为后面的实验做了铺垫。