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 3 月 1 日	

实 验 名 称　　实验 2 分词

一. 实验目的

了解分词（尤其是中文文本分词）的相关知识、算法及评价方式，学习并实现正向最大匹配算法和反向最大匹配算法与分词性能评价算法，并尝试与 Jieba 库的分词结果进行比较。

二. 实验内容

任务：使用正向最大匹配算法和字典文件（corpus.dict），对语料（corpus.sentence）进行分词，将分词的结果输出到文件 corpus.out 中；对比 corpus.answer 和 corpus.out，输出算法的 P/R/F 指标。

输出：

- 一个 corpus.out 文件（格式参照 corpus.answer）
- P/R/F 指标（格式类似于：Precision=36/100=36.00%）

语料数据格式：语料文本都为 utf-8 编码。

加分项：

1. 探索正向最大匹配算法和反向最大匹配算法哪个方法更适合中文分词？为什么（要实现反向最大匹配算法，通过实验数据来说明）？
2. 试学习使用 Python 第三方分词工具 jieba 进行分词，对比你实现的算法跟 jieba 分词的性能差异。

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。使用 Jest 编写测试代码。

（一）实验步骤

1. 在`./src`目录下创建`utils.ts`文件，用于导出若干辅助函数。

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


...
```

2. 创建第一个函数`readCorpusDict()`，它接受词典文件路径，先从第一行读取最大词长度，再读取可能出现的词汇的列表，并返回这两个结果。

这里使用了 Node.js 环境中的`node:fs/promises`包读取文件。代码看起来应当比较清楚。

```
/**
 * Read the corpus dictionary.
 * @param path The path of the corpus dictionary.
 */
export const readCorpusDict = async (
  path: string,
): Promise<{
  possibleWords: string[];
  maxWordLength: number;
}> => {
  const content = await fs.readFile(path, 'utf-8');
  const lines = content.split('\n');

  const maxWordLength = +lines[0].split('\t', 2)[1];
  const possibleWords = lines.slice(1);

  return { possibleWords, maxWordLength };
};
```

3. 再创建两个函数函数`readCorpusSource()`和`readCorpusProcessed()`。这两个函数都接受一个文件路径，分别读取 corpus.sentence.txt（未分词的源文件）和 corpus.answer.txt（人工分词的文件）。其中，前者返回去除了所有空白字符的**字符串**，而后者返回按空白字符分割后**再**去除空字符串的**字符串数组**（注意这里的区分）。

```
/**
 * Read the corpus source sentences.
 * @param path The path of the corpus source.
 */
export const readCorpusSource = async (path: string): Promise<string> => {
  const content = await fs.readFile(path, 'utf-8');
  return content.replace(/\s+/g, '');
};

/**
 * Read the corpus processed sentences.
 * @param path The path of the corpus processed.
 */
export const readCorpusProcessed = async (path: string): Promise<string[]> => {
  const content = await fs.readFile(path, 'utf-8');
  return content.split(/\s+/).filter((s) => s);
};
```

4. 创建函数`saveCorpusOutput()`。它接受**未分词的源文件的路径**、**要保存到的文件路径**和**作为分词结果的字符串数组**，然后保存结果。

之所以**要传入一个未分词的源文件路径**，是因为在分词后，字符串数组**丢失**了换行符所在位置的信息，因此需要根据未分词的源数据文件**重新插入换行符**。

```
/**
 * Save the segmentation results to a file,
 * separated by spaces, and add new lines according to the source.
 * @param sourcePath The path of the corpus source.
 * @param outputPath The path of the output file.
 * @param result The segmentation result.
 */
export const saveCorpusOutput = async (
  sourcePath: string,
  outputPath: string,
  result: string[],
) => {
  const sourceContent = await fs.readFile(sourcePath, 'utf-8');
  const sourceLines = sourceContent.split(/\r?\n/);
  const sourceLineLengths = sourceLines.map((line) => line.length);

  let output = '';
  let lineIdx = 0;
  let charIdx = 0;
  for (const word of result) {
    output += word;
    charIdx += word.length;

    if (charIdx === sourceLineLengths[lineIdx]) {
      output += '\n';
      lineIdx++;
      charIdx = 0;
    } else {
      output += ' ';
    }
  }

  await fs.writeFile(outputPath, output);
};
```

这段代码稍有些复杂，但逻辑如上所述，仅仅是根据未分词的源数据文件在相应的位置插入换行符而已。

5. 然后正式开始编写**前向最大匹配算法**的实现，即函数`forwardMaximumMatching()`：

```
/**
 * Use the forward maximum matching algorithm to segment.
 * @param text The text to be segmented.
 * @param possibleWords All words in the dictionary.
 * @param maxWordLength The length of the longest word in the dictionary.
 */
export const forwardMaximumMatching = (
  text: string,
```

```
  possibleWords: string[],
  maxWordLength: number,
): string[] => {
  const result: string[] = [];

  let index = 0; // The index of the current character
  while (index < text.length) {
    let length = maxWordLength; // The length of the current word
    let word = text.slice(index, index + length); // The current word

    // If the current word is not in the dictionary.
    while (length > 1 && !possibleWords.includes(word)) {
      length--;
      word = text.slice(index, index + length);
    }

    result.push(word);
    index += length;
  }

  return result;
};
```

这段代码的逻辑完全按照标准的前向最大匹配算法定义来写。对于不熟悉 JavaScript 的读者，这里可以说明一下数组的`Array#slice`方法同 Python 中的列表切片用法一致。除此之外，这里应当没有什么值得困惑的地方。

6. 同理，编写反向最大匹配算法的实现，即`backwardMaximumMatching()`函数：

```
/**
 * Use the backward maximum matching algorithm to segment.
 * @param text The text to be segmented.
 * @param possibleWords All words in the dictionary.
 * @param maxWordLength The length of the longest word in the dictionary.
 */
export const backwardMaximumMatching = (
  text: string,
  possibleWords: string[],
  maxWordLength: number,
): string[] => {
  const result: string[] = [];

  let index = text.length; // The index of the current character
  while (index > 0) {
    let length = Math.min(index, maxWordLength); // The length of the current word
    let word = text.slice(index - length, index); // the current word

    // If the current word is not in the dictionary.
```

```
    while (length > 1 && !possibleWords.includes(word)) {
      length--;
      word = text.slice(index - length, index);
    }

    result.unshift(word);
    index -= word.length;
  }

  return result;
};
```

这里的代码几乎与前向最大匹配算法完全一致，几乎只是把原来的“+”改成了“-”，并且从向数组末尾添加字符串变成了向数组首部添加字符串。

但是，很重要的一点是，**这里不再简单使用`maxWordLength`作为每次匹配的初始词组长度，而是使用了`Math.min(index, maxWordLength)`**。这是由于当反向最大匹配算法匹配到开头时，会出现`index – maxWordLength`小于 0 的情况，而且由于这里的`Array#slice`方法允许负值索引，因此很容易错误地将字符串中最后的几个字符添加到开头，所以这里必须进行处理。

7. 在完成两个分词算法后，编写`evaluate()`函数用于**评价分词结果**。该函数接受分词结果`result`和人工分词结果`answer`，并输出对应的 P/R/F 值。

```
/**
 * Evaluate the segmentation result (P / R / F1).
 * @param result Segmentation result array.
 * @param answer Manual segmentation result array.
 */
export const evaluate = (
  result: string[],
  answer: string[],
): {
  precisionFraction: [number, number];
  precision: number;
  recallFraction: [number, number];
  recall: number;
  f1: number;
} => {
  const forwardIdx = (words: string[], wordIdx: number, charIdx: number) => {
    if (charIdx < words[wordIdx].length - 1) {
      return [wordIdx, charIdx + 1];
    } else {
      return [wordIdx + 1, 0];
    }
  };

  const isCharAligned = (
    resultWords: string[],
```

```
    answerWords: string[],
    resultWordIdx: number,
    answerWordIdx: number,
    resultCharIdx: number,
    answerCharIdx: number,
  ) => {
    return (
      resultCharIdx === 0 &&
      answerCharIdx === 0 &&
      resultWords[resultWordIdx][0] === answerWords[answerWordIdx][0]
    );
  };

  return (() => {
    let resultWordIdx = 0;
    let answerWordIdx = 0;
    let resultCharIdx = 0;
    let answerCharIdx = 0;

    let correct = 0;

    while (resultWordIdx < result.length && answerWordIdx < answer.length) {
      if (result[resultWordIdx] === answer[answerWordIdx]) {
        // If the current word is correct, forward word index, and reset char index.
        correct++;
        resultWordIdx++;
        answerWordIdx++;
        resultCharIdx = 0;
        answerCharIdx = 0;
      } else {
        // Forward the index until the first character of the word is aligned.
        do {
          [resultWordIdx, resultCharIdx] = forwardIdx(
            result,
            resultWordIdx,
            resultCharIdx,
          );
          [answerWordIdx, answerCharIdx] = forwardIdx(
            answer,
            answerWordIdx,
            answerCharIdx,
          );
        } while (
          resultWordIdx < result.length &&
          answerWordIdx < answer.length &&
          !isCharAligned(
```

```
                result,
                answer,
                resultWordIdx,
                answerWordIdx,
                resultCharIdx,
                answerCharIdx,
              )
            );
          }
        }

        const precisionFraction: [number, number] = [correct, result.length];
        const precision = correct / result.length;
        const recallFraction: [number, number] = [correct, answer.length];
        const recall = correct / answer.length;
        const f1 = (2 * precision * recall) / (precision + recall);

        return { precisionFraction, precision, recallFraction, recall, f1 };
      })();
    };
```

这里的代码比较长，很可能令人费解，这里稍加解释。

首先解释这段代码的大致逻辑。我们定义四个变量 resultWordIdx、answerWordIdx、resultCharIdx 和 answerCharIdx，分别表示当前 result 数组中所选择**字符串**的下标，当前 answer 数组中所选择**字符串**的下标，当前 result 数组中**所选择字符串的**当前所选**字符**的下标，当前 answer 数组中**所选择字符串的**当前所选**字符**的下标。同时，还定义一个变量 correct，用来表示当前判断为匹配的词的数量。

一开始的时候，分别读取两个数组的第一个**字符串**，**假设**现在这两个**字符串**相同，这里的 correct++，表示相等词数量加 1，然后两个 wordIdx 加 1，进入下面的两个字符串判断。

假设这个时候，两个字符串不相等了，那么进入 else 分支，让**字符**（注意不是字符串）不断加 1（即两个 charIdx 同时加 1，如果某个 charIdx 加 1 时大于了当前字符串的长度，则 wordIdx 加 1 并且 charIdx 归零），直到两个字符**对齐**。

这里这么定义**对齐**：**对齐**指两个字符的 charIdx 都是 0（换言之，都在一个词组的开头），并且这两个字符相等。因为显而易见，只有当两个词组的第一个字符相等时，才有可能这两个词组相等（注意，这里必须要判断这两个 charIdx 是否为 0，否则有可能出现虽然两个字符相等，但两个 charIdx 不同，显然这两个词组不同）。

这里使用了 do { … } while { … }循环的目的是让这个 charIdx 加 1 的行为至少执行一次。因为在上面的例子中，最开始进入 else 分支时，两个词组的第一个字符显然是相等的，这时若使用 while { … }，就会发现 charIdx 没有加 1，会陷入死循环。这里不想用定义一个 flag 等不太优雅的方式解决问题，所以使用了一个 do { … } while { … }循环。

为什么要这么复杂？因为我们要通过字符每次**都**前进 1 来严格保证字符的对齐。否则，如果出现类似于“中国人民”和“中国”“人民”这样的情况，就可能导致中间出现意外的、字符未对齐的情况，那么后面很多本应该正确匹配的词都会因为字符没对齐而被判断为不匹配。

考虑下面这种情况：“中国人民”“解放”“军”“万岁”和“中国”“人民”“解放军”“万岁”。首先我们判断“中国人民”和“中国”不匹配，然后程序不断 charIdx 直到对齐的字符“解”，因为在这里第一个字符相等，然后 charIdx 也都是 0。但是，在下一轮循环中，发现“解放”不等于“解放军”，因此继续到下一个对齐的字符“万”，此时发现两个词都是“万岁”，因此判断相等。

现在，就很容易解释代码的实现了。首先在函数中通过闭包定义了两个辅助函数`forwardIdx()`和`isCharAligned()`，分别负责上面所说的“字符加 1”和“判断字符是否对齐”。然后在主要部分中通过循环统计正确词的数量。最后返回准确率的分子分母、准确率、召回率的分子分母、召回率以及 F.

这里的实现可能有不少冗余的地方，但通过每次只令字符增 1 **严格保证**了不会出现意外情况，因此该实现应当是准确的。

8. 除了这些算法实现之外，为了后面主函数的编写方便起见，我们编写一个辅助高阶函数 logged，用来简单地封装之后需要的打印日志的功能。

```
/**
 * Log the execution time of a function.
 * @param message The name of the function.
 * @param fn The function to be executed.
 */
export const logged = <T extends (...args: any[]) => any>(
  message: string,
  fn: T,
): T => {
  return ((...args) => {
    const startTime = Date.now();
    const result = fn(...args);
    if (result instanceof Promise) {
      return result
        .then((res) => {
          console.log(`${message} in ${Date.now() - startTime}ms.`);
          return res;
        })
        .catch((err) => {
          throw err;
        });
    }
    console.log(`${message} in ${Date.now() - startTime}ms.`);
    return result;
  }) as T;
};
```

这里的实现稍显复杂，并且为了同时兼容异步和同步函数使用了一些有些令人费解的技巧和类型定义。但是该函数与本实验报告的核心无关，因此不做解释。

8. 然后，在`./tests`目录下新建`utils.test.ts`，使用 Jest 为这些函数编写相应的测试。经测试全部通过，并且测试覆盖率为 100%。

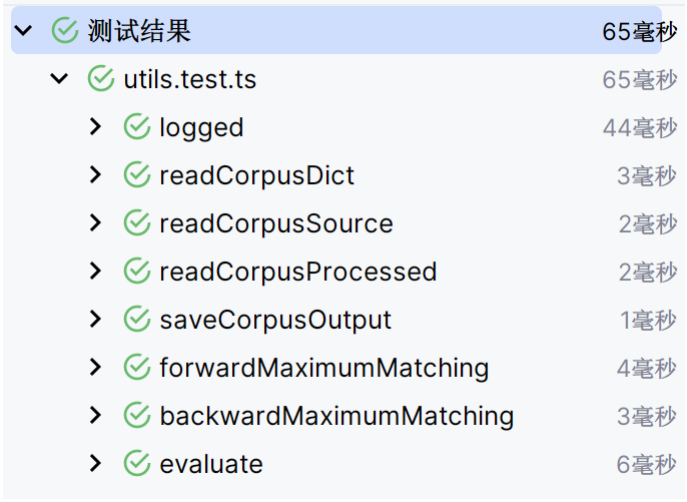

图 1 测试结果

图 2 测试覆盖率

下为测试代码。

```
import fs from 'node:fs/promises'
import {
  backwardMaximumMatching,
  evaluate,
  forwardMaximumMatching,
  logged,
  readCorpusDict,
  readCorpusProcessed,
  readCorpusSource,
  saveCorpusOutput
} from '../src/utils'

const POSSIBLE_WORDS = [
  '一',
  '一下',
  '下',
  '人',
  '人生',
  '人生大事',
```


教务处制

```
    '和',
    '和尚',
    '大',
    '大事',
    '好',
    '好好',
    '好好考虑',
    '尚',
    '尚未',
    '应',
    '应该',
    '未',
    '未结婚',
    '的',
    '结婚',
    '考',
    '考虑',
    '该',
    '都'
  ]
  const MAX_WORD_LENGTH = 4

  describe('logged', () => {
    beforeAll(() => {
      jest.spyOn(console, 'log').mockImplementation()
    })

    afterAll(() => {
      jest.clearAllMocks()
    })

    it('should log the execution time of a function', () => {
      const fn = jest.fn()
      logged('test', fn)()
      expect(console.log).toBeCalledWith(expect.stringMatching(/test in \d+ms./))
    })

    it('should not modify the return value of a normal function', () => {
      const fn = jest.fn().mockReturnValue(42)
      const result = logged('test', fn)()
      expect(result).toBe(42)
    })

    it('should not modify the return value of an async function', async () => {
      const fn = jest.fn().mockResolvedValue(42)
      const result = await logged('test', fn)()
```

```
    expect(result).toBe(42)
  })

  it('should throw the error of an async function', async () => {
    const fn = jest.fn().mockRejectedValue(new Error('test'))
    await expect(logged('test', fn)()).rejects.toThrow('test')
  })
})

describe('readCorpusDict', () => {
  beforeAll(() => {
    jest.spyOn(fs, 'readFile').mockImplementation(async () => {
      return '3\t10\nfoo\nbar\nbaz'
    })
  })

  afterAll(() => {
    jest.clearAllMocks()
  })

  it('should read corpus dict', async () => {
    const { possibleWords, maxWordLength } = await readCorpusDict('')

    expect(possibleWords).toEqual(['foo', 'bar', 'baz'])
    expect(maxWordLength).toBe(10)
  })
})

describe('readCorpusSource', () => {
  beforeAll(() => {
    jest.spyOn(fs, 'readFile').mockResolvedValue('我 是\n 中国  人')
  })

  afterAll(() => {
    jest.clearAllMocks()
  })

  it('should read corpus source', async () => {
    const content = await readCorpusSource('')

    expect(content).toBe('我是中国人')
  })
})

describe('readCorpusProcessed', () => {
  beforeAll(() => {
```

```
    jest.spyOn(fs, 'readFile').mockResolvedValue('我 是\n 中国  人')
  })

  afterAll(() => {
    jest.clearAllMocks()
  })

  it('should read processed corpus', async () => {
    const content = await readCorpusProcessed('')

    expect(content).toEqual(['我', '是', '中国', '人'])
  })
})

describe('saveCorpusOutput', () => {
  beforeAll(() => {
    jest.spyOn(fs, 'readFile').mockResolvedValue('我是\n 中国人')
    jest.spyOn(fs, 'writeFile').mockImplementation(async () => {})
  })

  afterAll(() => {
    jest.clearAllMocks()
  })

  it('should save processed output', async () => {
    await saveCorpusOutput('', '', ['我是', '中', '国人'])

    expect(fs.writeFile).toBeCalledWith(
      '',
      expect.stringMatching(/\s*我是\n 中  国人\s*/)
    )
  })
})

describe('forwardMaximumMatching', () => {
  it('should segment words', async () => {
    const words = forwardMaximumMatching(
      '结婚的和尚未结婚的都应该好好考虑一下人生大事',
      POSSIBLE_WORDS,
      MAX_WORD_LENGTH
    )

    expect(words).toEqual([
      '结婚',
      '的',
```

```
      '和尚',
      '未结婚',
      '的',
      '都',
      '应该',
      '好好考虑',
      '一下',
      '人生大事'
    ])
  })
})

describe('backwardMaximumMatching', () => {
  it('should segment words', async () => {
    const words = backwardMaximumMatching(
      '结婚的和尚未结婚的都应该好好考虑一下人生大事',
      POSSIBLE_WORDS,
      MAX_WORD_LENGTH
    )

    expect(words).toEqual([
      '结婚',
      '的',
      '和尚',
      '未结婚',
      '的',
      '都',
      '应该',
      '好好考虑',
      '一下',
      '人生大事'
    ])
  })
})

describe('evaluate', () => {
  it('should evaluate segmentation', () => {
    const answer = [
      '结婚',
      '的',
      '和',
      '尚未',
      '结婚',
      '的',
      '都',
```

```
        '应该',
        '好好',
        '考虑',
        '一下',
        '人生',
        '大事'
      ]
      const result = [
        '结婚',
        '的',
        '和尚',
        '未结婚',
        '的',
        '都',
        '应该',
        '好好考虑',
        '一下',
        '人生大事'
      ]
      const { precision, recall, f1 } = evaluate(result, answer)

      expect(Math.fround(precision).toFixed(4)).toBe('0.6000')
      expect(Math.fround(recall).toFixed(4)).toBe('0.4615')
      expect(Math.fround(f1).toFixed(4)).toBe('0.5217')
    })
  })
```

测试代码整体上比较简单。值得注意的是这里使用 Jest 的 Mock 模拟了 Node.js 与文件读取相关的几个函数，其他应当不用过多解释。

8. 在`./src/`目录下编写主函数`main.ts`。首先我们导入`./src/utils`中的几个函数，并且定义几个常量代码。代码如下：

```
import jieba from '@node-rs/jieba';
import {
  backwardMaximumMatching,
  evaluate,
  forwardMaximumMatching,
  logged,
  readCorpusDict,
  readCorpusProcessed,
  readCorpusSource,
  saveCorpusOutput,
} from './utils.js';

const CORPUS_DICT_PATH = 'data/corpus.dict.txt';
const CORPUS_SOURCE_PATH = 'data/corpus.sentence.txt';
const CORPUS_PROCESSED_PATH = 'data/corpus.answer.txt';
```

```
const CORPUS_OUTPUT_PATH = 'data/corpus.out.txt';
const CORPUS_OUTPUT_BACKWARD_PATH = 'data/corpus.out.backward.txt';
const CORPUS_OUTPUT_JIEBA_PATH = 'data/corpus.out.jieba.txt';

...
```

在这里，几个常量依次表示词典文件的路径、未分词的源数据文件的路径、人工分词文件的路径、正向最大匹配算法分词结果的保存路径、反向最大匹配算法分词结果的保存路径、使用 jieba 库分词结果的保存路径。

值得注意的是，这里使用了`@node-rs/jieba`库，这是 Python jieba 库的 Rust 移植版本，然后提供了 JavaScript 环境的接口，因此这里可以直接在 Node.js 环境中调用，并且由于使用 Rust 编写因而性能很高。

然后编写一个辅助函数`loadData()`.

```
const loadData = async () => {
  // Load the dictionary.
  const { possibleWords, maxWordLength } = await logged(
    'Dictionary loaded',
    readCorpusDict,
  )(CORPUS_DICT_PATH);

  // Load the source sentences.
  const source = await logged(
    'Source loaded',
    readCorpusSource,
  )(CORPUS_SOURCE_PATH);

  // Load the processed sentences.
  const answer = await logged(
    'Processed loaded',
    readCorpusProcessed,
  )(CORPUS_PROCESSED_PATH);

  return { possibleWords, maxWordLength, source, answer };
};
```

该函数逻辑非常简单，就是分别调用了几个读取数据的函数把词典、未分词的字符串和人工分词的结果读进来，并且打印日志，没什么特别的。

然后再编写一个辅助函数`logEvaluation()`，该函数的作用就是把`evaluate()`函数输出的结果打印出来，没什么需要解释的。

```
const logEvaluation = ({
  precisionFraction,
  precision,
  recallFraction,
  recall,
  f1,
}: {
```

```
  precisionFraction: [number, number];
  precision: number;
  recallFraction: [number, number];
  recall: number;
  f1: number;
}) => {
  const toPercentage = (num: number) => `${(num * 100).toFixed(4)}%`;

  console.log(
    `Precision = ${precisionFraction[0]} / ${
      precisionFraction[1]
    } = ${toPercentage(precision)}`,
  );
  console.log(
    `Recall = ${recallFraction[0]} / ${recallFraction[1]} = ${toPercentage(
      recall,
    )}`,
  );
  console.log(`F1 = ${toPercentage(f1)}`);
};
```

最后，编写主函数`main()`并且执行它，本实验就完成了。

```
const main = async () => {
  // Load the data.
  const { possibleWords, maxWordLength, source, answer } = await loadData();

  console.log();

  // Use forward maximum matching algorithm to segment.
  const result = logged(
    'Forward maximum matching finished',
    forwardMaximumMatching,
  )(source, possibleWords, maxWordLength);
  // Evaluate the segmentation result.
  logEvaluation(evaluate(result, answer));
  // Save the segmentation result.
  await logged('Result saved', saveCorpusOutput)(
    CORPUS_SOURCE_PATH,
    CORPUS_OUTPUT_PATH,
    result,
  );

  console.log();

  // Use backward maximum matching algorithm to segment.
  const result2 = logged(
```

```
      'Backward maximum matching finished',
      backwardMaximumMatching,
    )(source, possibleWords, maxWordLength);
    // Evaluate the segmentation result.
    logEvaluation(evaluate(result2, answer));
    // Save the segmentation result.
    await logged('Result saved', saveCorpusOutput)(
      CORPUS_SOURCE_PATH,
      CORPUS_OUTPUT_BACKWARD_PATH,
      result2,
    );

    console.log();

    // Use Jieba to evaluate the segmentation result.
    const result3 = logged('Jieba finished', jieba.cut)(source);
    // Evaluate the segmentation result.
    logEvaluation(evaluate(result3, answer));
    // Save the segmentation result.
    await logged('Result saved', saveCorpusOutput)(
      CORPUS_SOURCE_PATH,
      CORPUS_OUTPUT_JIEBA_PATH,
      result3,
    );
  };

  await main();
```

主函数的逻辑很简单：分别进行正向最大匹配、反向最大匹配和 jieba 分词，然后对每一个都打印执行时间和输出结果的评价信息，并且将结果保存到文件中。

（二）实验结果

运行`npm run dev`，使用 ts-node 执行`./src/main.ts`，输出如下。

```
Dictionary loaded in 6ms.
Source loaded in 10ms.
Processed loaded in 16ms.

Forward maximum matching finished in 2402ms.
Precision = 20263 / 20397 = 99.3430%
Recall = 20263 / 20454 = 99.0662%
F1 = 99.2044%
Result saved in 41ms.

Backward maximum matching finished in 2596ms.
Precision = 20273 / 20404 = 99.3580%
Recall = 20273 / 20454 = 99.1151%
F1 = 99.2364%
Result saved in 31ms.

Jieba finished in 650ms.
Precision = 16114 / 19768 = 81.5156%
Recall = 16114 / 20454 = 78.7817%
F1 = 80.1253%
```

图 3 输出结果

四. 实验总结

1. 相比于正向最大匹配算法，反向最大匹配算法经实验推测可能更适合中文分词（两者的准确率 Precision 相近，而 Recall 和 F 值反向最大匹配算法均高于正向最大匹配算法）。猜测原因是由于中文的语义往往由后面的词决定，例如“硕士研究生产”，用正向最大匹配可能分出“硕士研究生/产”，而反向最大匹配则很可能分出“硕士/研究/生产”，可以看到正向最大匹配没能很好地理解语义，把“生产”这个词给错误地切分了。这个例子说明了**反向最大匹配**能够更好地处理中文的语义结构，因为**中文的修饰成分往往在前面，而核心成分在后面**，这与英文正好相反。

2. 本实验也使用了 jieba 进行分词作为对比。对比起这里的原生 Node.js 实现，jieba 的速度大约为 5~10 倍。这是很合理的性能差异。这里使用的`@node-rs/jieba`是 Python jieba 库的 Rust 移植版本，相较 Python jieba 库还要更快，比我们的原生实现更快是很正常的。

事实上，我们可能把上面算法中的`Array#includes`方法改成一个集（Set），这样就能使用 O(1)的时间复杂度来判断是否存在于词典中，这样就只有 O(n)的复杂度了。

```ts
/**
 * Use the forward maximum matching algorithm to segment.
 * @param text The text to be segmented.
 * @param possibleWords All words in the dictionary.
 * @param maxWordLength The length of the longest word in the dictionary.
 */
export const forwardMaximumMatching = (
  text: string,
  possibleWords: string[],
  maxWordLength: number,
): string[] => {
```

```
    const result: string[] = [];
    const possibleWordsSet = new Set(possibleWords);

    let index = 0; // The index of the current character
    while (index < text.length) {
      let length = maxWordLength; // The length of the current word
      let word = text.slice(index, index + length); // The current word

      // If the current word is not in the dictionary.
      while (length > 1 && !possibleWordsSet.has(word)) {
        length--;
        word = text.slice(index, index + length);
      }

      result.push(word);
      index += length;
    }

    return result;
  };
```

现在我们再跑一遍`./src/main.ts`，可以看到**我们的实现甚至比 jieba 还要快**：

```
Forward maximum matching finished in 46ms.
Precision = 20263 / 20397 = 99.3430%
Recall = 20263 / 20454 = 99.0662%
F1 = 99.2044%
Result saved in 39ms.

Backward maximum matching finished in 143ms.
Precision = 20273 / 20404 = 99.3580%
Recall = 20273 / 20454 = 99.1151%
F1 = 99.2364%
Result saved in 26ms.

Jieba finished in 546ms.
Precision = 16114 / 19768 = 81.5156%
Recall = 16114 / 20454 = 78.7817%
F1 = 80.1253%
```

图 4 使用 Set 加速后的实验结果

为什么会这样？jieba 使用了两种算法来进行分词，一种基于前缀词典，一种基于 HMM，前者的时间复杂度是 O(n)，后者是 $O(n^2)$，前者的 n 取决于文本长度，后者取决于未登录词的长度。也就是说，jieba 所使用的算法在时间复杂度上与我们没有太大的差异，并且与词典大小无关，但使用了较复杂的算法，因此在常量时间上应当需要花更多时间。此外，由于我们是在 Node.js 环境中调用该库，中间还要转一层接口，也会花一些时间，这也要考虑在其中。

并且，jieba 的分词准确率也比我们的要低。这应当只是由于题中所给的字典过分适合题目的情景，因此分词结果非常准确。而 jieba 使用了一个通用的大词典，分词准确率在 80%左右是一个正常的水

平。虽然 jieba 在这个场景下准确率低于我们的实现，但在通用性上显然要高于我们的实现。

3. 在本次实验中，使用 Node.js 环境编写了正向最大匹配算法和反向最大匹配算法这两个分词算法的实现，实现了根据外部词库实现中文分词、评价结果，并保存结果到相应路径的功能。并且，使用 jieba 库也进行了分词，并将其结果与我们的实现进行对比。

通过本次实验，了解了 NLP 领域分词的基本知识，并且实践了两种基本的分词算法和 P/R/F 评价函数的实现。