西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

《数据挖掘》大作业

——多文档自动摘要

学　　院: 自动化学院

专　　业: 机器人工程

姓　　名: 张镇韬

学　　号: 2020302419

得　　分:

2023 年 5 月

目录

一、 引言 3
1.1 多文档自动摘要的重要性和应用 3
1.2 目标和研究问题 3
二、 相关工作 4
2.1 自动文本摘要技术 4
2.1.1 抽取式摘要 4
2.1.2 生成式摘要 4
2.1.3 混合摘要方法 4
2.1.4 多文档自动摘要技术 4
2.2 评价指标 5
2.2.1 ROUGE 5
2.2.2 BLEU 5
2.2.3 METEOR 5
三、 数据集和预处理 5
3.1 DUC 2004 任务 2 数据集描述 5
3.1.1 数据集组织结构 5
3.1.2 参考摘要 6
3.1.3 数据集的用途 6
3.2 数据预处理方法 6
3.2.1 句子切分 6
3.2.2 去除停用词 6
3.2.3 词干化 6
四、 方法实现 7
4.1 Baseline 方法 7
4.2 基于 TF-IDF 和余弦相似度的摘要生成方法 7
4.3 基于聚类的余弦相似度文本摘要生成方法 9
4.4 基于句向量相似度的文本摘要生成方法 12
4.5 基于 ROUGE 的评价方法 12
4.6 基于 ROUGE 评价方法的排序算法 14
五、 结果展示 14
六、 结论 17
附录 19

一、 引言

1.1 多文档自动摘要的重要性和应用

随着信息技术的快速发展，互联网上的文本数据量呈现出爆炸式增长。每天都有大量的新闻、报告、论文等文本信息产生，人们面临着如何从海量数据中提取关键信息的挑战。多文档自动摘要（Multi-document Automatic Summarization）应运而生，旨在从多篇相关文档中自动生成简短、准确和连贯的摘要，使得用户能够在较短的时间内获取关键信息。

多文档自动摘要在实际应用中具有广泛的重要性和价值，以下列举了一些应用场景：

- 新闻聚合：在新闻聚合应用中，多文档自动摘要可以帮助用户从众多来源的新闻报道中快速了解主要内容和观点，节省阅读时间。
- 文献检索与研究：学术研究人员可以利用多文档自动摘要技术从大量相关论文中提取关键信息，加快文献调研和理解的速度，从而提高研究效率。
- 情报分析：在商业和政治领域，多文档自动摘要可以帮助分析师从大量文本数据中提取关键信息，以便更好地进行竞争分析和决策。
- 在线教育：教师和学生可以使用多文档自动摘要工具从多个课程材料和在线资源中提取关键概念和知识点，有助于提高教学和学习效果。
- 社交媒体分析：多文档自动摘要可以用于分析社交媒体上的舆论动态，从大量用户评论和帖子中提炼关键信息，为企业和政府提供有价值的市场和舆情分析。

因此，多文档自动摘要技术在各个领域具有广泛的应用前景，对于提高信息获取效率和辅助决策具有重要价值。随着自然语言处理和人工智能技术的不断发展，多文档自动摘要的性能和准确性将得到进一步提高，为用户提供更好的信息服务。

1.2 目标和研究问题

本研究的目标是探究和评估多文档自动摘要技术在处理大量文本数据时的有效性和可行性。为了实现这一目标，我们将关注以下研究问题：

1. 如何从多篇相关文档中提取关键信息并生成简短、准确和连贯的摘要？

为了解决这一问题，我们需要研究现有的多文档自动摘要方法，包括抽取式摘要和生成式摘要。抽取式摘要通过从原始文档中选择关键句子或短语来构建摘要，而生成式摘要则利用深度学习模型生成新的句子。我们将分析各种方法的优缺点，并尝试找到适用于特定应用场景的最佳策略。

2. 如何评估多文档自动摘要的质量？

评估自动摘要的质量是一个关键问题，因为它可以为我们提供关于算法性能的重要反馈。本研究将探讨现有的评估方法，如 ROUGE 评分和人工评估。我们将分析这些方法的优缺点，并探索在不同场景下使用哪种评估方法最为合适。

3. 如何克服多文档自动摘要中的挑战？

多文档自动摘要面临着一些挑战，如处理冗余信息、保持摘要的连贯性和可读性以及处理多样性的文档集。本研究将分析这些挑战，并探讨如何通过算法改进和优化来克服这些问题。

通过对以上研究问题的深入探讨，本研究旨在为多文档自动摘要技术的发展和应用提供有益的指导，从而提高人们在面对大量文本数据时的信息获取效率。

二、 相关工作

2.1 自动文本摘要技术

自动文本摘要技术是一种自然语言处理任务，其目标是从原始文本中提取关键信息并生成简短、准确和连贯的摘要。自动文本摘要技术主要分为以下几类：

2.1.1 抽取式摘要

抽取式摘要方法通过从原始文档中提取关键句子或短语来生成摘要。这些方法通常依赖于诸如句子重要性、句子相似性和文本结构等特征。抽取式摘要技术的主要优点是生成的摘要通常具有较高的可读性和准确性，因为摘要中的内容直接来自于原文。然而，由于摘要是通过选择原始文本中的部分内容生成的，因此可能存在连贯性和流畅性方面的问题。抽取式摘要的典型方法包括：

基于句子重要性的方法：如 TF-IDF、TextRank 等；

基于句子相似性的方法：如 K-means 聚类、最大边缘相关性等；

基于文本结构的方法：如首尾句子抽取、基于段落的抽取等。

2.1.2 生成式摘要

与抽取式摘要不同，生成式摘要方法通过学习原始文本的语义表示来生成新的句子。生成式摘要通常利用深度学习模型，如循环神经网络（RNN）、长短时记忆网络（LSTM）和 Transformer 等。生成式摘要的主要优点是生成的摘要可以更好地保持连贯性和流畅性，因为模型可以生成新的句子，而不仅仅是从原始文本中选择内容。然而，生成式摘要可能面临着准确性和可读性方面的挑战，因为生成的摘要可能包含与原文不符的信息或语法错误。生成式摘要的典型方法包括：

序列到序列（seq2seq）模型：如基于 LSTM 的编码器-解码器架构；

注意力机制：如带有注意力的编码器-解码器模型；

预训练语言模型：如 BERT、GPT 等。

2.1.3 混合摘要方法

为了充分利用抽取式摘要和生成式摘要的优点，一些研究采用了混合摘要方法。这些方法结合了抽取和生成技术，以生成更准确、连贯和可读的摘要。混合摘要方法可以采用多种策略，如：

双阶段方法：首先使用抽取式摘要方法从原始文本中提取关键句子，然后将这些句子作为输入，使用生成式摘要方法生成摘要；

基于模型的集成：在一个统一的框架下，同时使用抽取式和生成式摘要模型，通过权重调整和优化来生成最终摘要；

联合训练：将抽取式摘要和生成式摘要任务作为多任务学习问题进行联合训练，以便在生成摘要时同时考虑两者的优点。

2.1.4 多文档自动摘要技术

在多文档自动摘要任务中，系统需要从一组相关的文档中生成一个综合摘要。多文档自动摘要面临的挑战包括处理不同文档中的重复信息、解决信息不一致和冲突以及维护摘要的连贯性。为了解决这些问题，多文档自动摘要方法通常采用以下策略：

文档预处理：将文档转换为统一的表示形式，以便于后续处理；

信息抽取和融合：从多个文档中抽取关键信息并进行融合，以消除冗余和解决不一致问题；

摘要生成：根据抽取和融合的信息，使用抽取式、生成式或混合摘要方法生成最终摘要。

多文档自动摘要技术在新闻汇总、事件分析、企业情报和学术研究等领域具有广泛的应用前景。为了评估多文档自动摘要系统的性能，研究人员通常采用基于 ROUGE 指标的评估方法，该方法通过比较系统生成的摘要与人工撰写的参考摘要之间的重叠来衡量摘要的质量。

2.2 评价指标

在自动文本摘要任务中，评估一个系统的性能至关重要。为了衡量生成摘要的质量，研究人员提出了一系列评价指标。以下是一些常用的评价指标：

2.2.1 ROUGE

ROUGE（Recall-Oriented Understudy for Gisting Evaluation）是一种广泛使用的自动文本摘要评价指标。它通过将系统生成的摘要与人工编写的参考摘要进行比较，计算它们之间的重叠程度。ROUGE 指标有多个变体，包括：

ROUGE-N：计算 n-gram（如单词或短语）在系统生成摘要和参考摘要之间的重叠。通常使用 ROUGE-1（单词重叠）和 ROUGE-2（二元短语重叠）进行评估。

ROUGE-L：使用最长公共子序列（LCS）方法计算摘要之间的重叠，以评估摘要的句子层面相似性。

ROUGE-S：计算基于跳跃的二元短语（skip-bigram）重叠，允许摘要中的词语以不同的顺序出现。

ROUGE-SU：结合 ROUGE-S 和单词单元（unigram）重叠，以评估摘要的更细粒度相似性。

ROUGE 指标通常报告召回率（recall）、准确率（precision）和 F1 分数（F1-score）。召回率表示参考摘要中有多少信息被系统生成的摘要所覆盖，准确率表示系统生成摘要中有多少信息是与参考摘要一致的，而 F1 分数则是召回率和准确率的调和平均值，用于综合评估摘要的质量。

2.2.2 BLEU

BLEU（Bilingual Evaluation Understudy）最初是为机器翻译任务设计的评价指标，但在自动文本摘要任务中也有一定的应用。与 ROUGE 类似，BLEU 指标通过计算系统生成摘要与参考摘要之间的 n-gram 重叠来评估摘要的质量。然而，与 ROUGE 不同的是，BLEU 通常只报告准确率（precision）而不是召回率（recall）。因此，在某些情况下，BLEU 指标可能会对过短的摘要产生偏好。

2.2.3 METEOR

METEOR（Metric for Evaluation of Translation with Explicit ORdering）是另一种机器翻译评价指标，也可用于自动文本摘要任务。METEOR 通过计算系统生成摘要与参考摘要之间的单词

三、 数据集和预处理

3.1 DUC 2004 任务 2 数据集描述

Document Understanding Conference (DUC) 是由美国国家标准与技术研究所（National Institute of Standards and Technology，NIST）组织的一系列会议，旨在推动自动文本摘要技术的发展。DUC 2004 任务 2 数据集是一个广泛使用的多文档摘要数据集，用于评估多文档自动摘要系统的性能。

3.1.1 数据集组织结构

DUC 2004 任务 2 数据集包含 50 个主题，每个主题下有 10 篇新闻文章，共计 500

篇文档。文档涵盖了广泛的主题，如政治、经济、社会和科技等。每篇文章都是从各种新闻来源收集的，如美联社（Associated Press），路透社（Reuters）和纽约时报（New York Times）等。

3.1.2 参考摘要

对于每个主题，数据集提供了 4 个由人类专家编写的参考摘要。这些摘要作为评估自动摘要系统生成摘要质量的基准。每个参考摘要的长度限制为 100 字，以模拟实际应用中对摘要长度的限制。参考摘要包含了文档集中的主要信息，旨在提供对原始文档的精确、简洁和有代表性的概述。

3.1.3 数据集的用途

DUC 2004 任务 2 数据集被广泛应用于自动摘要技术的研究和评估。研究人员使用该数据集来训练和测试各种多文档摘要方法，以及评估不同摘要系统之间的性能差异。通过将系统生成的摘要与人类编写的参考摘要进行比较，可以量化地评估自动摘要系统在内容覆盖、非冗余、连贯性和可读性等方面的表现。

3.2 数据预处理方法

在对多文档自动摘要系统进行研究时，数据预处理是一个关键步骤。预处理方法的目的是对原始文本进行清洗、格式化和简化，以便更好地应用于后续的摘要生成任务。

我们采用以下预处理步骤：

3.2.1 句子切分

将原始文档切分成独立的句子。这是通过使用 nltk 库中的 sent_tokenize 函数实现的。nltk（Natural Language Toolkit）是一个广泛使用的自然语言处理（NLP）库，它为文本处理、分类、实体识别、词性标注、句法分析等任务提供了丰富的工具和资源。代码如下：

```
1. import nltk
2. from nltk import sent_tokenize
3. sentences = nltk.sent_tokenize(content)
```

3.2.2 去除停用词

停用词是指那些在文本中频繁出现，但对于文本的主题和含义贡献较小的词。去除停用词可以减少数据维度，并帮助突显重要信息。我们使用 nltk 库中提供的英文停用词列表，并通过列表解析去除句子中的停用词。

```
1. import nltk
2. from nltk.corpus import stopwords
3. nltk.download('stopwords')
4. stopwords_list = set(stopwords.words("english"))
```

3.2.3 词干化

词干化是一种将词汇还原为其基本形式（词干）的过程。这可以帮助减少数据维度，使不同形式的同一词汇在后续处理中被视为相同。我们使用 The Porter Stemming Algorithm 进行词干化，通过 nltk 库中的`PorterStemmer`类实现：

```
1. from nltk.stem import PorterStemmer
```

```
2. stemmer = PorterStemmer()
```

四、 方法实现

4.1 Baseline 方法

在本节中，我们将介绍用于多文档自动摘要任务的 Baseline 方法。该方法非常简单，仅从每个文档中提取第一句话作为摘要的一部分。将所有这些句子连接在一起，就得到了针对整个文档集的摘要。下面详细解释了实现此方法的过程（具体代码见附录 1）：

1. 导入所需的库和模块：os、re、time 和 nltk。
2. 定义一个名为 extract_first_sentence 的函数，该函数接受一个文档文件路径作为参数。函数的任务是从文件中提取第一句话。

a. 读取文件内容。

b. 使用正则表达式提取包含在<TEXT>和</TEXT>标签之间的文本。

c. 去除<TEXT>和</TEXT>标签以及换行符。

d. 使用 nltk.sent_tokenize 将文本分割成句子。

e. 返回第一句话。

3. 设置源文档路径和输出摘要文件夹路径。
4. 遍历源文档路径下的所有子文件夹。

a. 对于每个子文件夹(即每个主题),创建一个空列表 baseline_summary_sentences。

b. 遍历子文件夹中的所有文件（即文档）。

c. 对于每个文件，调用 extract_first_sentence 函数并将返回的句子添加到 baseline_summary_sentences 列表中。

d. 将 baseline_summary_sentences 列表中的所有句子连接起来，形成摘要。

e. 将摘要写入输出文件夹中的相应文件。

5. 输出处理进度和总用时。

通过这种简单的 Baseline 方法，我们可以生成针对每个主题的摘要。虽然这种方法可能无法捕捉到文档集中的所有重要信息，但它可以作为一个基准，用于评估其他更复杂的摘要方法的性能。

4.2 基于 TF-IDF 和余弦相似度的摘要生成方法

在本节中，我们将介绍一种基于 TF-IDF 和余弦相似度的摘要生成方法。此方法可以作为我们后续改进的基准方法。以下是算法的具体步骤：

1. 使用 TF-IDF 算法将 DUC 文本数据转化为数值型数据。

TF-IDF（Term Frequency-Inverse Document Frequency）是一种用于信息检索和文本挖掘的常用加权技术。它的主要思想是：如果某个词在一篇文章中出现的频率高，并且在其他文章中很少出现，则认为这个词具有很好的类别区分能力。TF-IDF 算法有两部分组成：词频（Term Frequency, TF）和逆文档频率（Inverse Document Frequency, IDF）。

词频（Term Frequency, TF）：指的是某一个给定的词语在该文件中出现的频率。这个数字通常会被归一化，以防止它偏向长的文章。计算公式为：

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d} \tag{1}$$

逆文档频率（Inverse Document Frequency, IDF）：如果包含词条的文件越少，则说

明词条具有很好的类别区分能力。某一特定词语的 IDF，可以由总文件数目除以包含该词语的文件的数目，再将得到的商取对数得到。即为：

$$\mathrm{IDF}(t,D) = \log \frac{\text{Total number of documents in } D}{\text{Number of documents containing term } t} \tag{2}$$

其中，t 代表词语，d 代表文档，D 代表文档集合。

因此。TF-IDF 的计算公式为：

$$\mathrm{TF-IDF}(t,\ d,\ D) = \mathrm{TF}(t,\ d) \cdot \mathrm{IDF}(t,\ D) \tag{3}$$

2. 构建句子-词语矩阵。

其中每一行表示一个句子，每一列表示一个词语，矩阵中的元素表示该词语的 TF-IDF 值。句子是通过它们的词频-逆文档频率（TF-IDF）值表示成向量的。每个句子的向量表示形式是一个一维数组，其长度等于词汇表中的词语数量。数组中的每个元素对应于词汇表中某个词语的 TF-IDF 值。

3. 使用余弦相似度构建句子相似度矩阵。

余弦相似度是一种常用的相似度度量方法，它通过计算两个向量间的夹角的余弦值来评估他们的相似度。假设我们有两个向量 A 和 B，则余弦相似度的计算公式为：

$$\mathrm{cosine-similarity}(A,B) = \frac{A \cdot B}{|A||B|} \tag{4}$$

4. 对句子进行排序。

将主题中的 10 个文档视为一个长句子。将每个短句子与长句子进行余弦相似度计算。按相似度分值从大到小对句子进行排序。

5. 控制冗余并生成摘要。

选择相似度最高的句子作为 s1。将分值第二高的句子与第一个句子进行相似度比较，只有当相似度小于某个阈值时，才将其放入摘要集合中。对于每个选定的摘要句子，需要与摘要集合中的所有句子进行相似度比较，只有当相似度小于某个阈值的句子才能选择进入摘要集合。重复此过程，直到摘要集合中的词语数达到 665 字节，算法终止。摘要集合中的句子即为计算机生成的摘要结果。

本算法的伪代码如下图所示（Python 代码见附录 2）：

Algorithm 1 TFIDF-CosineSimilaritySummarizer

Require: docs_path, threshold, byte_limit
Ensure: Summary for each topic

```
for folder in os.listdir(docs_path) do
  if os.path.isdir(folder_path) then
    original_topic_sentences = []
    processed_topic_sentences = []
    for file in os.listdir(folder_path) do
      file_path = os.path.join(folder_path, file)
      original_sentences, processed_sentences = preprocess(file_path)
      original_topic_sentences.extend(original_sentences)
      processed_topic_sentences.extend(processed_sentences)
    end for
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(processed_topic_sentences)
    sentence_similarity_matrix = cosine_similarity(X)
    sentence_similarity_graph = {i: np.where(sentence_similarity_matrix[i]
    >= threshold)[0] for i in range(len(processed_topic_sentences))}
    summary = generate_summary(sentence_similarity_graph, original_topic_sentences, processed_topic_sentences, vectorizer, threshold, byte_limit)
    output_file = os.path.join(output_folder, f"{folder}-TFIDF-CosineSimilaritySummarizer.txt")
    Save summary to output_file
  end if
end for
```

图 1 基于 TF-IDF 和余弦相似度的摘要生成方法的伪代码

4.3 基于聚类的余弦相似度文本摘要生成方法

在这个基准方法中，我们使用基于聚类的余弦相似度文本摘要器（Cluster-Based Cosine Similarity Summarizer）对 TF-IDF 和余弦相似度算法进行了改进。此方法包括以下步骤：

1. 将预处理后的句子聚类成 k 个簇，其中每个簇看作一个句子。

我们使用 TF-IDF 向量化器将预处理后的句子转换为向量表示 X，之后计算簇的数量 n_clusters = int(sqrt(len(processed_sentences)))，这里采用 $\sqrt{n}$ 作为簇的数量。最后使用 KMeans 算法对向量 X 进行聚类，得到簇标签 cluster_labels 和簇中心 cluster_centers。

KMeans 算法的目标是将数据点分成 k 个簇，使得每个簇内的数据点到其对应簇中心的平方距离之和最小。假设我们有 n 个数据点 $x_1, x_2, ..., x_n$，KMeans 算法试图找到 k 个簇中心 $c_1, c_2, ..., c_k$ 使得以下目标函数最小化：

$$J(c_1, \cdots, c_k) = \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - c_i||^2 \tag{5}$$

其中 S_i 表示属于簇中心 c_i 的数据点集合。在我们的方法中，KMeans 算法通过迭

代优化过程来实现，具体过程如下：

i. 随机选择k个数据点作为初始簇中心。

ii. 将每个数据点分配给距离最近的簇中心，从而形成k个簇。

iii. 更新每个簇的中心为簇内数据点的平均值。

iv. 重复步骤 ii 和 iii，直到簇中心不再发生显著变化或达到最大迭代次数。

在聚类过程中，我们使用欧几里得距离来度量数据点之间的距离。欧几里得距离定义如下：

$$d(x,\ y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{6}$$

其中x和y是具有n个特征的数据点。

2. 计算每个簇c_i和簇中包含的句子s_j之间的余弦相似度；计算每个簇c_i和整个主题之间的余弦相似度。

3. 对每个簇c_i中的句子s_j按相似度分值大小进行排序；对每个簇c_i按余弦相似度大小进行排序；从排序最高的簇到排序最低的簇，依次选取每个簇的排序最高句子；一轮过后，再从排序最高的簇到排序最低的簇，依次选取每个簇的排序次高句子。

伪代码如下（Python 代码见附录 3）：

Algorithm 2 Cluster-Based Cosine Similarity Summarizer

Require: file_path, byte_limit
Ensure: summary

```
import os, re, math, nltk, numpy, TfidfVectorizer, cosine_similarity, KMeans
function preprocess(file_path):
{Read file content and extract sentences}
sentences ← extract_sentences(file_path)
{Preprocess sentences: remove stopwords, perform stemming}
original_sentences, processed_sentences ← preprocess_sentences(sentences)
return original_sentences, processed_sentences
function generate_improved_summary(original_sentences, processed_sentences, vectorizer, byte_limit):
{Vectorize processed sentences}
X ← vectorizer.fit_transform(processed_sentences)
{Perform KMeans clustering}
n_clusters, kmeans ← perform_kmeans(X)
{Calculate cosine similarity between cluster centers and sentences}
cluster_sentence_similarities, cluster_center_similarities ← calculate_cosine_similarities(kmeans, original_sentences, processed_sentences, vectorizer)
{Select sentences for summary based on cosine similarity and byte limit}
summary ← select_sentences(cluster_center_similarities, cluster_sentence_similarities, byte_limit)
return summary
{Main program}
docs_path, output_folder, byte_limit ← set_input_output_paths_and_byte_limit()
topic_count, start_time ← 0, time.time()
for folder in os.listdir(docs_path) do
  if os.path.isdir(os.path.join(docs_path, folder)) then
    {Preprocess all files in the folder}
    original_topic_sentences, processed_topic_sentences ← preprocess_all_files(folder)
    vectorizer ← TfidfVectorizer()
    {Generate summary for the topic}
    summary ← generate_improved_summary(original_topic_sentences, processed_topic_sentences, vectorizer, byte_limit)
    {Write summary to output file}
    write_summary_to_file(folder, output_folder, summary)
    topic_count ← topic_count + 1
    {Print progress}
    if topic_count % 10 == 0 then
      print_progress(topic_count, start_time)
    end if
  end if
end for{Print total elapsed time}
print_total_elapsed_time(start_time)
```

图 2 基于聚类的余弦相似度文本摘要生成方法的伪代码

4.4 基于句向量相似度的文本摘要生成方法

该方法通过使用句向量表示（Sentence2Vec）和余弦相似度来计算句子之间的相似度。基于相似度矩阵，我们可以对句子进行排序并选择最具代表性的句子来生成摘要。整个过程包括以下步骤：

1. 数据预处理：将文档切分为句子并将句子表示为句子向量（Sentence2Vec）。

在这个方法中，我们使用了 sentence_transformers 库和预训练的 distilbert-base-nli-mean-tokens 模型来将句子表示为句子向量。这个模型是基于 DistilBERT 的句子嵌入模型，能够捕捉句子的语义信息。

2. 计算句子的相似度。
3. 构建相似度矩阵。
4. 对句子排序。
5. 生成摘要。

伪代码如下（Python 代码见附录 4）：

Algorithm 3 Sentence2Vec Similarity Summarizer

```
Require: file_path, threshold, byte_limit
function PREPROCESS(file_path)
    read content from file_path
    EXTRACT text_block from content
    SPLIT text_block into original_sentences
    return original_sentences, processed_sentences

function GENERATE_SUMMARY(sentence_similarity_matrix, original_sentences, processed_sentences, threshold, byte_limit)
    CONSTRUCT sentence_similarity_graph from sentence_similarity_matrix
    SORT sentences by similarity graph length
    SELECT non-similar sentences based on threshold and byte_limit
    return summary

PREPARE necessary libraries and models

for each folder in docs_path do
    PREPROCESS and COLLECT sentences from files in folder
    COMPUTE sentence embeddings using sentence_model
    COMPUTE sentence_similarity_matrix
    GENERATE_SUMMARY(sentence_similarity_matrix, original_sentences, processed_sentences, threshold, byte_limit)
    SAVE summary to output_file
end for
```

图 3 基于句向量相似度的文本摘要生成方法的伪代码

4.5 基于 ROUGE 的评价方法

如 2.2 节所述，自动文摘领域已经发展了几十年，期间产生了许多评测方法。其中，基于召回率的评测标准（Recall-Oriented Understudy for Gisting Evaluation，简称 ROUGE）被认为是最可靠和最稳定的评测方法。自 2004 年以来，ROUGE 一直被 DUC（文档理解会议）作为官方评测方法。

为了评估自动摘要的质量，我们编写了一个 Python 程序（代码见附录 5），计算生成的摘要与参考摘要之间的 ROUGE 得分。我们使用 rouge 库来计算 ROUGE 得分。

程序主要包括以下步骤：

1. 定义一个函数 calculate_rouge_scores 来计算摘要之间的 ROUGE 得分。

我们主要计算了三个 ROUGE 指标：ROUGE-1，ROUGE-2 和 ROUGE-L。这些指标是通过比较生成的摘要和参考摘要之间的相似性来计算的：

ROUGE-N：计算生成的摘要和参考摘要之间 N 元词（n-gram）的匹配率。ROUGE-N 召回率（R）和准确率（P）的计算公式分别为：

$$R_n = \frac{\sum_{S \in \text{Reference Summaries}} \sum_{\text{gram}_n \in S} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{S \in \text{Reference Summaries}} \sum_{\text{gram}_n \in S} \text{Count}(\text{gram}_n)} \tag{7}$$

$$P_n = \frac{\sum_{\text{gram}_n \in \text{Generated Summary}} \text{Count}_{match}(\text{gram}_n)}{\sum_{\text{gram}_n \in \text{Generated Summary}} \text{Count}(\text{gram}_n)} \tag{8}$$

其中，$\text{Count}_{match}(\text{gram}_n)$表示参考摘要和生成摘要中相匹配的 n-gram 数量，$\text{Count}(\text{gram}_n)$表示 n-gram 的总数量。接着，我们可以计算 F-值（F）：

$$F_n = \frac{2 \times P_n \times R_n}{P_n + R_n} \tag{9}$$

ROUGE-L：计算生成的摘要和参考摘要之间的最长公共子序列（LCS）匹配。ROUGE-L 的召回率（R）、准确率（P）和 F-值（F）的计算公式分别为：

$$R_L = \frac{\sum_{S \in \text{Reference Summaries}} \text{LCS}(S, \text{Generated Summary})}{\sum_{S \in \text{Reference Summaries}} |S|} \tag{10}$$

$$P_L = \frac{\text{LCS}(\text{Generated Summary}, S_{\text{best}})}{|\text{Generated Summary}|} \tag{11}$$

$$F_L = \frac{2 \times P_L \times R_L}{P_L + R_L} \tag{12}$$

其中，$\$\text{LCS}(S, \text{Generated Summary})$表示生成摘要与参考摘要 S 之间的最长公共子序列的长度，S_{best}表示与生成摘要具有最长 LCS 的参考摘要，$|S|$表示参考摘要 S 的长度，|Generated Summary| 表示生成摘要的长度。

在这段代码中，我们使用 Python 的 rouge 库来计算这些指标，然后计算所有生成摘要的平均 ROUGE 得分。

2. 定义一个函数 read_file 来读取文件内容。
3. 读取生成的摘要和参考摘要。
4. 对每个生成的摘要，使用 calculate_rouge_scores 函数计算其与参考摘要之间的 ROUGE 得分。
5. 将所有生成摘要的 ROUGE 得分求平均值。
6. 计算此方法的得分S_o（Overlap score）。

在这段代码中，我们首先计算了每个生成摘要的ROUGE-1，ROUGE-2和ROUGE-L的F-值。然后，我们计算了平均值。最后，我们使用权重对这些平均F-值求和，得到Overlap score。具体来说，我们计算Overlap score的公式如下：

$$S_o = w_1 \times \overline{F}_1 + w_2 \times \overline{F}_2 + w_3 \times \overline{F}_L \tag{13}$$

其中，w_1、w_2和w_3分别是ROUGE-1、ROUGE-2和ROUGE-L的权重。在这段代码中，我们为这三个指标分配了相同的权重$\frac{1}{3}$，因此：

$$S_o = \frac{1}{3}\overline{F}_1 + \frac{1}{3}\overline{F}_2 + \frac{1}{3}\overline{F}_L \tag{14}$$

这为我们提供了一个统一的衡量标准，反映了生成摘要与参考摘要之间的相似性。权重可以根据实际需求进行调整。

4.6 基于ROUGE评价方法的排序算法

我们将介绍基于ROUGE评价方法的排序算法，这是4.5提到的算法的改进和应用。该算法利用ROUGE评价指标对生成的摘要进行评分，并根据平均重叠分数（Overlap score）对参与比赛的35个团队进行排名。

代码的主要步骤如下（具体代码见附录6）：

1. 导入所需的库和模块。
2. 定义read_file函数，用于读取文件内容。
3. 定义calculate_rouge_scores函数，计算生成摘要与参考摘要之间的ROUGE得分。
4. 定义calculate_overlap_score函数，计算生成摘要的平均重叠分数。
5. 读取输入文件夹中的文件，将文件内容保存到字典中，按照主题ID和团队ID进行分组。
6. 针对每个主题，计算参考摘要。
7. 计算每个团队在每个主题上的摘要的平均重叠分数。
8. 计算每个团队的总平均重叠分数。
9. 根据总平均重叠分数对团队进行排序，并输出结果。

五、 结果展示

如上一章节所述，我们编写了相关的代码如下图所示：

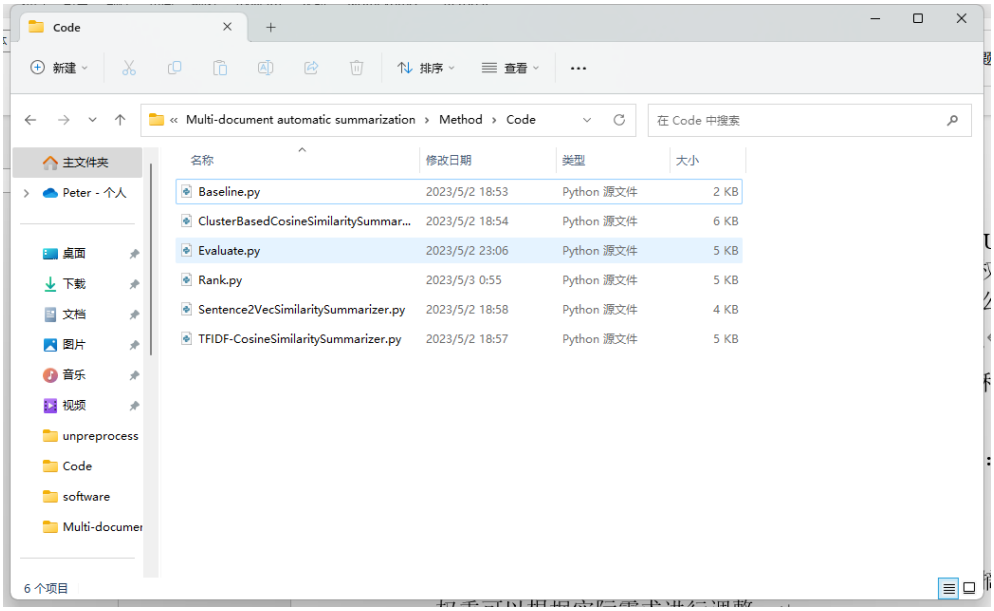

图 4 代码结构

分别运行 Baseline、TFIDF-CosineSimilaritySummarizer、ClusterBasedCosineSimilaritySummarizer、Sentence2VecSimilaritySummarizer 等代码，结果如下图所示（这里以 Baseline 为例，其余代码类似）：

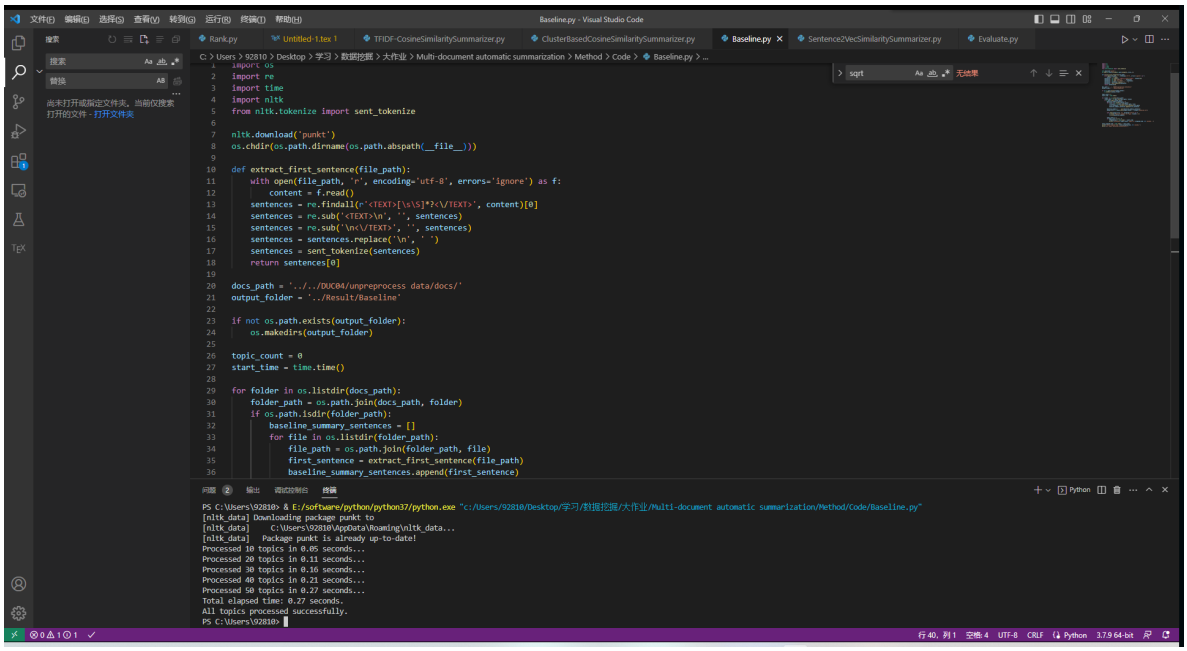

图 5 代码运行结果

比较各个代码的运行时间，可得到下表：

表 1 运行时间

方法	Baseline	TFIDF	Cluster	Sentence2Vec
运行时间（s）	0.27	4.05	21.64	106.76

可见，由于 Baseline 最简单，只需要提取句子，因此计算最快。而最慢的是 Sentence2Vec，需要通过神经网络模型转换成向量，因此计算最慢，平均 2 秒以上才能生成一篇摘要。

产生的摘要结构和内容如下图所示（仍以 Baseline 为例）：

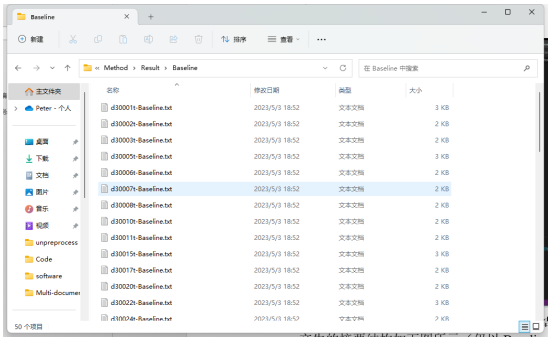

图 6 结果文件夹结构

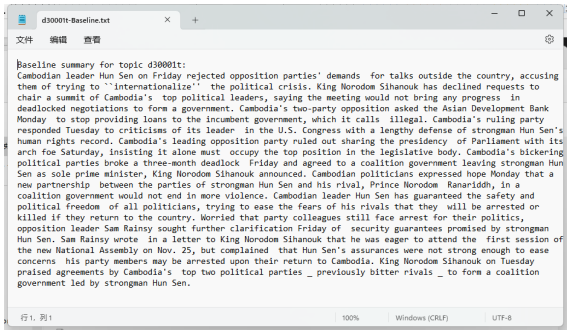

图 7 生成的摘要

接着，使用 4.5 节中介绍的算法对各个方法计算评价指标，运行结果如下图所示（仍以 Baseline 为例）：

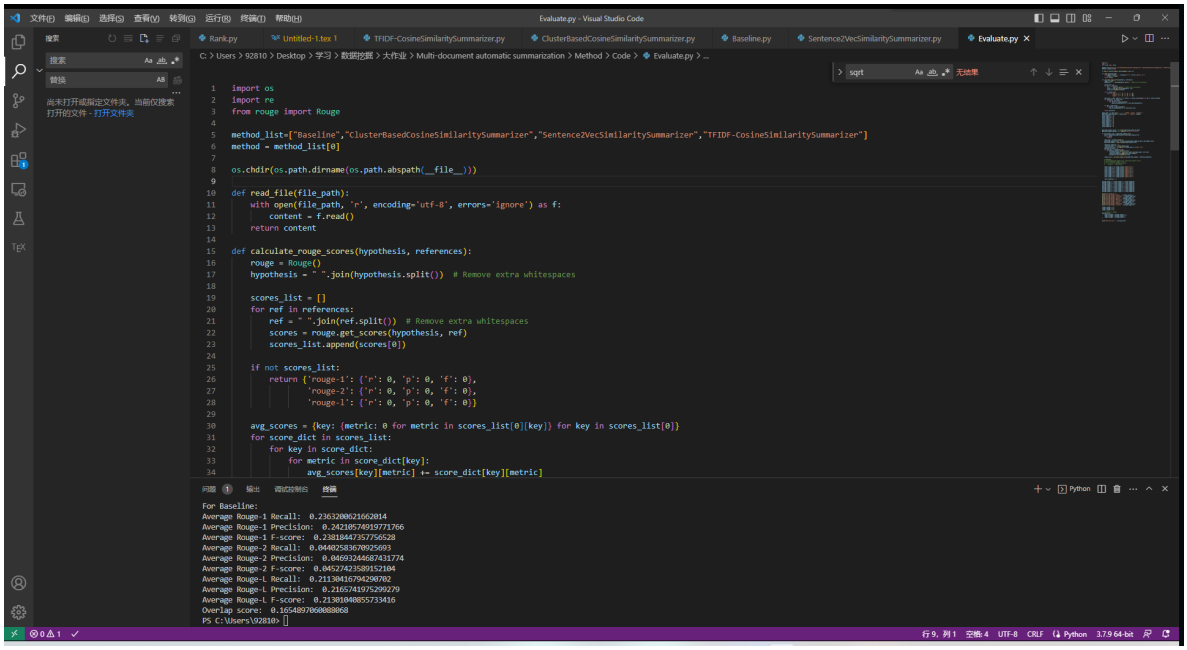

图 8 计算评价指标

整理得到下表：

表 2 各个方法的评价指标

	Baseline	TFIDF	Cluster	Sentence2Vec
Average Rouge-1 Recall	0.2195	0.2090	0.2363	0.4188
Average Rouge-1 Precision	0.1981	0.2203	0.2421	0.2071
Average Rouge-1 F-score	0.2079	0.2134	0.2382	0.2748
Average Rouge-2 Recall	0.0335	0.0316	0.0440	0.1196
Average Rouge-2 Precision	0.0315	0.0316	0.0469	0.0506
Average Rouge-2 F-score	0.0325	0.0331	0.0453	0.0705
Average Rouge-L Recall	0.1974	0.1821	0.2113	0.3803
Average Rouge-L Precision	0.1782	0.1921	0.2166	0.1879
Average Rouge-L F-score	0.1870	0.1860	0.2130	0.2494
Overlap score	0.1425	0.1442	0.1655	0.1982

由上表可以看出，Baseline 的 overlap 得分最低，而 TFIDF 的 overlap 略高，但高的不是很明显。这并不意味着我们的方法改进不大，而是因为 Baseline 抽取了所有文档的第一句话，篇幅较长，而 TFIDF 受到篇幅限制，只有 Baseline 的三分之一篇幅，因此得分提高不是很明显。而 Cluster 和 Sentence2Vec 的得分则有较为明显的提高，这也是符合我们预期的，因为对句子含义的把握越来越准确，相应的，生成摘要花费的时间也越来越长。

之后，我们运行 4.6 节中提到的代码，计算 35 支队伍的 overlap score，可以得到最高分为 0.2026 分，最低分为 0.1047 分，平均分为 0.1714 分（完整结果见附录 7）。

若把我们的方法放入其中一起进行排序，则 Baseline 和 TFIDF 算法可以排到第 33 名，Cluster 算法可以排到第 21 名，Sentence2Vec 算法可以排到第 4 名，证明我们的算法还是很有竞争力的。

六、 结论

本报告详细介绍了多文档自动摘要的研究背景、目标和研究问题。在相关工作部分，我们回顾了当前自动文本摘要技术的发展，包括抽取式摘要、生成式摘要、混合摘要方法以及多文档自动摘要技术。我们还讨论了用于评估自动摘要的主要指标，如 ROUGE、BLEU 和 METEOR。

在数据集和预处理部分，我们描述了 DUC 2004 任务 2 数据集的组织结构、参考摘要和用途。我们还讨论了数据预处理方法，包括句子切分、去除停用词和词干化。

在方法实现部分，本报告介绍了五种不同的文本摘要生成方法：Baseline 方法、基于 TF-IDF 和余弦相似度的摘要生成方法、基于聚类的余弦相似度文本摘要生成方法以及基于句向量相似度的文本摘要生成方法。此外，我们使用 ROUGE 指标编写评价摘要质量的算法。每种方法都进行了详细的实现说明和分析。

在结果展示部分，我们比较了各种方法在 DUC 2004 任务 2 数据集上的表现。结果表明，基于聚类的余弦相似度文本摘要生成方法和基于句向量相似度的文本摘要生成方法在某些情况下具有较好的性能。此外，基于 ROUGE 评价方法的排序算法可以帮助我们找出最佳表现的摘要生成方法。我们的三种方法在 35 支参赛队中最好可以排到第 4 名。

总之，本报告对多文档自动摘要的研究方法和技术进行了全面的探讨。我们的研究成果可以为实际应用提供指导，并为未来研究提供有价值的参考。然而，自动摘要技术仍有很大的提升空间，尤其是在生成式摘要和混合摘要方法方面。未来的研究可以在这些方向上继续深入探究，以实现更高质量的文本摘要。

附录

附录 1

介绍：Python 编写 Baseline 方法

```
1. import os
2. import re
3. import time
4. import nltk
5. from nltk.tokenize import sent_tokenize
6.
7. nltk.download('punkt')
8. os.chdir(os.path.dirname(os.path.abspath(__file__)))
9.
10. def extract_first_sentence(file_path):
11.     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
12.         content = f.read()
13.     sentences = re.findall(r'<TEXT>[\s\S]*?<\/TEXT>', content)[0]
14.     sentences = re.sub('<TEXT>\n', '', sentences)
15.     sentences = re.sub('\n<\/TEXT>', '', sentences)
16.     sentences = sentences.replace('\n', ' ')
17.     sentences = sent_tokenize(sentences)
18.     return sentences[0]
19.
20. docs_path = '../../DUC04/unpreprocess data/docs/'
21. output_folder = '../Result/Baseline'
22.
23. if not os.path.exists(output_folder):
24.     os.makedirs(output_folder)
25.
26. topic_count = 0
27. start_time = time.time()
28.
29. for folder in os.listdir(docs_path):
30.     folder_path = os.path.join(docs_path, folder)
31.     if os.path.isdir(folder_path):
32.         baseline_summary_sentences = []
33.         for file in os.listdir(folder_path):
34.             file_path = os.path.join(folder_path, file)
35.             first_sentence = extract_first_sentence(file_path)
36.             baseline_summary_sentences.append(first_sentence)
37.
38.         baseline_summary = ' '.join(baseline_summary_sentences)
39.         output_file = os.path.join(output_folder, f"{folder}-
    Baseline.txt")
```

```
40.
41.         with open(output_file, "w", encoding="utf-8") as f:
42.             f.write(f"Baseline summary for topic {folder}:\n")
43.             f.write(baseline_summary)
44.
45.         topic_count += 1
46.         if topic_count % 10 == 0:
47.             elapsed_time = time.time() - start_time
48.             print(f"Processed {topic_count} topics in {elapsed_time:.2f}
    seconds...")
49.
50. total_elapsed_time = time.time() - start_time
51. print(f"Total elapsed time: {total_elapsed_time:.2f} seconds.")
52. print("All topics processed successfully.")
```

附录 2

介绍：Python 编写 TFIDF-CosineSimilaritySummarizer 方法

```
1.  import os
2.  import re
3.  import nltk
4.  import time
5.  import numpy as np
6.  from nltk.corpus import stopwords
7.  from nltk.stem import PorterStemmer
8.  from sklearn.feature_extraction.text import TfidfVectorizer
9.  from sklearn.metrics.pairwise import cosine_similarity
10.
11. os.chdir(os.path.dirname(os.path.abspath(__file__)))
12. docs_path = '../../DUC04/unpreprocess data/docs/'
13. threshold = 0.5
14. byte_limit = 665
15. nltk.download('punkt')
16. nltk.download('stopwords')
17.
18. def preprocess(file_path):
19.     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
20.         content = f.read()
21.     sentences = re.findall(r'<TEXT>[\s\S]*?<\/TEXT>', content)[0]
22.     sentences = re.sub('<TEXT>\n', '', sentences)
23.     sentences = re.sub('\n<\/TEXT>', '', sentences)
```

```
    sentences = sentences.replace('\n', ' ')
    original_sentences = nltk.sent_tokenize(sentences)
    stopwords_list = set(stopwords.words("english"))
    stemmer = PorterStemmer()
    processed_sentences = []
    for sentence in original_sentences:
        words = nltk.word_tokenize(sentence)
        words = [word.lower() for word in words if word.isalpha()]
        words = [word for word in words if word not in stopwords_list]
        words = [stemmer.stem(word) for word in words]
        processed_sentence = ' '.join(words)
        processed_sentences.append(processed_sentence)
    return original_sentences, processed_sentences


def generate_summary(sentence_similarity_graph, original_sentences,
processed_sentences, vectorizer, threshold, byte_limit):
    selected_sentences = []
    sorted_sentences = sorted(sentence_similarity_graph, key=lambda k:
len(sentence_similarity_graph[k]), reverse=True)

    for index in sorted_sentences:
        candidate_sentence = processed_sentences[index]
        candidate_vector = vectorizer.transform([candidate_sentence])
        is_similar = False

        for selected_sentence in selected_sentences:
            selected_index = original_sentences.index(selected_sentence)
            selected_vector =
vectorizer.transform([processed_sentences[selected_index]])
            similarity = cosine_similarity(candidate_vector,
selected_vector)
            if similarity >= threshold:
                is_similar = True
                break

        if not is_similar:
            selected_sentences.append(original_sentences[index])
            current_bytes = sum([len(sentence.encode('utf-8')) for sentence
in selected_sentences])

            if current_bytes > byte_limit:
                selected_sentences.pop()
```

```
            break

    summary = ' '.join(selected_sentences)
    return summary

output_folder = '../Result/TFIDF-CosineSimilaritySummarizer'
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

topic_count = 0
start_time = time.time()

for folder in os.listdir(docs_path):
    folder_path = os.path.join(docs_path, folder)
    if os.path.isdir(folder_path):
        original_topic_sentences = []
        processed_topic_sentences = []
        for file in os.listdir(folder_path):
            file_path = os.path.join(folder_path, file)
            original_sentences, processed_sentences = preprocess(file_path)
            original_topic_sentences.extend(original_sentences)
            processed_topic_sentences.extend(processed_sentences)

        vectorizer = TfidfVectorizer()
        X = vectorizer.fit_transform(processed_topic_sentences)
        sentence_similarity_matrix = cosine_similarity(X)
        sentence_similarity_graph = {i:
    np.where(sentence_similarity_matrix[i] >= threshold)[0] for i in
    range(len(processed_topic_sentences))}

        summary = generate_summary(sentence_similarity_graph,
    original_topic_sentences, processed_topic_sentences, vectorizer,
    threshold, byte_limit)

        output_file = os.path.join(output_folder, f"{folder}-TFIDF-
    CosineSimilaritySummarizer.txt")
        with open(output_file, "w", encoding="utf-8") as f:
            f.write(f"Summary for topic {folder}:\n")
            f.write(summary)

        topic_count += 1
        if topic_count % 10 == 0:
            elapsed_time = time.time() - start_time
```

```
            print(f"Processed {topic_count} topics in {elapsed_time:.2f} seconds...")

total_elapsed_time = time.time() - start_time
print(f"Total elapsed time: {total_elapsed_time:.2f} seconds.")
print("All topics processed successfully.")
```

附录 3

介绍：Python 编写 Cluster-Based Cosine Similarity Summarizer 方法

```
import os
import re
import math
import nltk
import time
import numpy as np
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.cluster import KMeans

os.chdir(os.path.dirname(os.path.abspath(__file__)))

nltk.download('punkt')
nltk.download('stopwords')

def preprocess(file_path):
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        content = f.read()
    sentences = re.findall(r'<TEXT>[\s\S]*?<\/TEXT>', content)[0]
    sentences = re.sub('<TEXT>\n', '', sentences)
    sentences = re.sub('\n<\/TEXT>', '', sentences)
    sentences = sentences.replace('\n', ' ')
    original_sentences = nltk.sent_tokenize(sentences)
    stopwords_list = set(stopwords.words("english"))
    stemmer = PorterStemmer()
    processed_sentences = []
    for sentence in original_sentences:
        words = nltk.word_tokenize(sentence)
        words = [word.lower() for word in words if word.isalpha()]
        words = [word for word in words if word not in stopwords_list]
        words = [stemmer.stem(word) for word in words]
```

```
        processed_sentence = ' '.join(words)
        processed_sentences.append(processed_sentence)
    return original_sentences, processed_sentences

def generate_improved_summary(original_sentences, processed_sentences, vectorizer, byte_limit):
    X = vectorizer.fit_transform(processed_sentences)
    n_clusters = int(math.sqrt(len(processed_sentences)))
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
    cluster_labels = kmeans.labels_
    cluster_centers = kmeans.cluster_centers_

    cluster_sentence_similarities = {}
    cluster_center_similarities = []
    for i in range(n_clusters):
        cluster_original_sentences = [original_sentences[j] for j in range(len(processed_sentences)) if cluster_labels[j] == i]
        cluster_processed_sentences = [processed_sentences[j] for j in range(len(processed_sentences)) if cluster_labels[j] == i]
        cluster_vectors = vectorizer.transform(cluster_processed_sentences)
        cluster_similarities = cosine_similarity(cluster_vectors, cluster_centers[i].reshape(1, -1)).flatten()
        cluster_sentence_similarities[i] = cluster_similarities
        cluster_center_similarity = np.mean(cluster_similarities)
        cluster_center_similarities.append((i, cluster_center_similarity))

    cluster_center_similarities.sort(key=lambda x: x[1], reverse=True)
    selected_sentences = []
    current_bytes = 0
    max_rounds = len(processed_sentences) // n_clusters + 1
    for _ in range(max_rounds):
        for cluster_index, _ in cluster_center_similarities:
            cluster_sentences = [(j, original_sentences[j]) for j in range(len(processed_sentences)) if cluster_labels[j] == cluster_index]
            cluster_sentence_similarities[cluster_index] = sorted(cluster_sentence_similarities[cluster_index], reverse=True)
            while len(cluster_sentence_similarities[cluster_index]) > 0:
                highest_similarity_index = np.argmax(cluster_sentence_similarities[cluster_index])
                selected_sentence = cluster_sentences[highest_similarity_index][1]
                selected_sentence_bytes = len(selected_sentence.encode('utf-8'))
```

```
            if current_bytes + selected_sentence_bytes <= byte_limit:
                selected_sentences.append(selected_sentence)
                current_bytes += selected_sentence_bytes
                break
            else:
                cluster_sentence_similarities[cluster_index] =
np.delete(cluster_sentence_similarities[cluster_index],
highest_similarity_index)
                cluster_sentences.pop(highest_similarity_index)
        if current_bytes >= byte_limit:
            break
    if current_bytes >= byte_limit:
        break

summary = ' '.join(selected_sentences)
return summary

docs_path = '../../DUC04/unpreprocess data/docs/'
output_folder = '../Result/ClusterBasedCosineSimilaritySummarizer'
byte_limit = 665

if not os.path.exists(output_folder):
    os.makedirs(output_folder)

topic_count = 0
start_time = time.time()

for folder in os.listdir(docs_path):
    folder_path = os.path.join(docs_path, folder)
    if os.path.isdir(folder_path):
        original_topic_sentences = []
        processed_topic_sentences = []
        for file in os.listdir(folder_path):
            file_path = os.path.join(folder_path, file)
            original_sentences, processed_sentences =
preprocess(file_path)
            original_topic_sentences.extend(original_sentences)
            processed_topic_sentences.extend(processed_sentences)

        vectorizer = TfidfVectorizer()
        summary = generate_improved_summary(original_topic_sentences,
processed_topic_sentences, vectorizer, byte_limit)

```

```
        output_file = os.path.join(output_folder, f"{folder}-ClusterBasedCosineSimilaritySummarizer.txt")
        with open(output_file, "w", encoding="utf-8") as f:
            f.write(f"Summary for topic {folder}:\n")
            f.write(summary)

        topic_count += 1
        if topic_count % 10 == 0:
            elapsed_time = time.time() - start_time
            print(f"Processed {topic_count} topics in {elapsed_time:.2f} seconds...")

total_elapsed_time = time.time() - start_time
print(f"Total elapsed time: {total_elapsed_time:.2f} seconds.")
print("All topics processed successfully.")
```

附录 4

介绍：Python 编写 Sentence2VecSimilaritySummarizer 方法

```
import os
import re
import numpy as np
import nltk
import time
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer

os.chdir(os.path.dirname(os.path.abspath(__file__)))
nltk.download('punkt')
nltk.download('stopwords')

def preprocess(file_path):
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        content = f.read()
    text_block = re.findall(r'<TEXT>[\s\S]*?<\/TEXT>', content)[0]
    text_block = re.sub('<TEXT>\n', '', text_block)
    text_block = re.sub('\n<\/TEXT>', '', text_block)
    original_sentences = nltk.sent_tokenize(text_block.replace('\n', ' '))

    # Apply preprocessing here as required
```

```
    processed_sentences = original_sentences

    return original_sentences, processed_sentences

def generate_summary(sentence_similarity_matrix, original_sentences, processed_sentences, threshold, byte_limit):
    sentence_similarity_graph = {i: np.where(sentence_similarity_matrix[i] >= threshold)[0] for i in range(len(processed_sentences))}
    sorted_sentences = sorted(sentence_similarity_graph, key=lambda k: len(sentence_similarity_graph[k]), reverse=True)

    selected_sentences = []
    for index in sorted_sentences:
        candidate_sentence = original_sentences[index]
        is_similar = False

        for selected_sentence in selected_sentences:
            similarity = sentence_similarity_matrix[index][selected_sentences.index(selected_sentence)]
            if similarity >= threshold:
                is_similar = True
                break

        if not is_similar:
            selected_sentences.append(candidate_sentence)
            current_bytes = sum([len(sentence.encode('utf-8')) for sentence in selected_sentences])

            if current_bytes > byte_limit:
                selected_sentences.pop()
                break

    summary = ' '.join(selected_sentences)
    return summary

docs_path = '../../DUC04/unpreprocess data/docs/'
output_folder = '../Result/Sentence2VecSimilaritySummarizer'
threshold = 0.5
byte_limit = 665

model_path = os.path.abspath("../distilbert-base-nli-mean-tokens")
sentence_model = SentenceTransformer(model_path)
```

```
61.
62. if not os.path.exists(output_folder):
63.     os.makedirs(output_folder)
64.
65. topic_count = 0
66. start_time = time.time()
67.
68. for folder in os.listdir(docs_path):
69.     folder_path = os.path.join(docs_path, folder)
70.     if os.path.isdir(folder_path):
71.         original_topic_sentences = []
72.         processed_topic_sentences = []
73.         for file in os.listdir(folder_path):
74.             file_path = os.path.join(folder_path, file)
75.             original_sentences, processed_sentences = preprocess(file_path)
76.             original_topic_sentences.extend(original_sentences)
77.             processed_topic_sentences.extend(processed_sentences)
78.
79.         sentence_embeddings =
    sentence_model.encode(processed_topic_sentences)
80.         sentence_similarity_matrix =
    cosine_similarity(sentence_embeddings)
81.
82.         summary = generate_summary(sentence_similarity_matrix,
    original_topic_sentences, processed_topic_sentences, threshold,
    byte_limit)
83.
84.         output_file = os.path.join(output_folder, f"{folder}-
    Sentence2VecSimilaritySummarizer.txt")
85.         with open(output_file, "w", encoding="utf-8") as f:
86.             f.write(f"Summary for topic {folder}:\n")
87.             f.write(summary)
88.
89.         topic_count += 1
90.         if topic_count % 10 == 0:
91.             elapsed_time = time.time() - start_time
92.             print(f"Processed {topic_count} topics in {elapsed_time:.2f}
    seconds...")
93.
94. total_elapsed_time = time.time() - start_time
95. print(f"Total elapsed time: {total_elapsed_time:.2f} seconds.")
96. print("All topics processed successfully.")
```

附录 5

介绍：Python 编写的 ROUGE 评估方法

```
import os
import re
from rouge import Rouge

method_list=["Baseline","ClusterBasedCosineSimilaritySummarizer","Sentence2VecSimilaritySummarizer","TFIDF-CosineSimilaritySummarizer"]
method = method_list[3]

os.chdir(os.path.dirname(os.path.abspath(__file__)))

def read_file(file_path):
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        content = f.read()
    return content

def calculate_rouge_scores(hypothesis, references):
    rouge = Rouge()
    hypothesis = " ".join(hypothesis.split())  # Remove extra whitespaces

    scores_list = []
    for ref in references:
        ref = " ".join(ref.split())  # Remove extra whitespaces
        scores = rouge.get_scores(hypothesis, ref)
        scores_list.append(scores[0])

    if not scores_list:
        return {'rouge-1': {'r': 0, 'p': 0, 'f': 0},
                'rouge-2': {'r': 0, 'p': 0, 'f': 0},
                'rouge-l': {'r': 0, 'p': 0, 'f': 0}}

    avg_scores = {key: {metric: 0 for metric in scores_list[0][key]} for key in scores_list[0]}
    for score_dict in scores_list:
        for key in score_dict:
            for metric in score_dict[key]:
                avg_scores[key][metric] += score_dict[key][metric]

    for key in avg_scores:
        for metric in avg_scores[key]:
            avg_scores[key][metric] /= len(scores_list)

```

```
    return avg_scores

model_path = os.path.join("..", "..", "DUC04", "model", "04model")
generated_summaries_path = os.path.join("..", "Result", method)
total_rouge_1_r = 0
total_rouge_1_p = 0
total_rouge_1_f = 0
total_rouge_2_r = 0
total_rouge_2_p = 0
total_rouge_2_f = 0
total_rouge_l_r = 0
total_rouge_l_p = 0
total_rouge_l_f = 0
total_summaries = 0


generated_summary_files = os.listdir(generated_summaries_path)
generated_summary_pattern = re.compile(r"d3(\d+)t-.*\.txt")

for gen_summary_file in generated_summary_files:
    match = generated_summary_pattern.match(gen_summary_file)
    if not match:
        continue

    topic_num = int(match.group(1))
    generated_summary_file = os.path.join(generated_summaries_path,
gen_summary_file)
    generated_summary = read_file(generated_summary_file)

    reference_summaries = []
    reference_files = os.listdir(model_path)
    reference_pattern = re.compile(f"D3{topic_num:04}.M.100.T.\\w")
    matched_ref_files = []
    for ref_file in reference_files:
        if reference_pattern.match(ref_file):
            reference_summary = read_file(os.path.join(model_path,
ref_file))
            reference_summaries.append(reference_summary)
            matched_ref_files.append(ref_file)

    rouge_scores = calculate_rouge_scores(generated_summary,
reference_summaries)

    #打印文件名
```

```
    # print(f"Generated summary file: {generated_summary_file}")
    # print("Reference summary files:")
    # for ref_file in matched_ref_files:
    #     print(f"  - {ref_file}")

    total_rouge_1_r += rouge_scores['rouge-1']['r']
    total_rouge_1_p += rouge_scores['rouge-1']['p']
    total_rouge_1_f += rouge_scores['rouge-1']['f']
    total_rouge_2_r += rouge_scores['rouge-2']['r']
    total_rouge_2_p += rouge_scores['rouge-2']['p']
    total_rouge_2_f += rouge_scores['rouge-2']['f']
    total_rouge_l_r += rouge_scores['rouge-l']['r']
    total_rouge_l_p += rouge_scores['rouge-l']['p']
    total_rouge_l_f += rouge_scores['rouge-l']['f']

    total_summaries += 1

average_rouge_1_r = total_rouge_1_r / total_summaries
average_rouge_1_p = total_rouge_1_p / total_summaries
average_rouge_1_f = total_rouge_1_f / total_summaries
average_rouge_2_r = total_rouge_2_r / total_summaries
average_rouge_2_p = total_rouge_2_p / total_summaries
average_rouge_2_f = total_rouge_2_f / total_summaries
average_rouge_l_r = total_rouge_l_r / total_summaries
average_rouge_l_p = total_rouge_l_p / total_summaries
average_rouge_l_f = total_rouge_l_f / total_summaries

print(f"For {method}:")
# print("Average Rouge-1 Recall: ", average_rouge_1_r)
# print("Average Rouge-1 Precision: ", average_rouge_1_p)
# print("Average Rouge-1 F-score: ", average_rouge_1_f)
# print("Average Rouge-2 Recall: ", average_rouge_2_r)
# print("Average Rouge-2 Precision: ", average_rouge_2_p)
# print("Average Rouge-2 F-score: ", average_rouge_2_f)
# print("Average Rouge-L Recall: ", average_rouge_l_r)
# print("Average Rouge-L Precision: ", average_rouge_l_p)
# print("Average Rouge-L F-score: ", average_rouge_l_f)

rouge_1_weight = 1/3
rouge_2_weight = 1/3
rouge_l_weight = 1/3

# 使用 F1 分数计算加权平均
overlap_score = (
```

```
    rouge_1_weight * average_rouge_1_f +
    rouge_2_weight * average_rouge_2_f +
    rouge_l_weight * average_rouge_l_f
)

print("Overlap score: ", overlap_score)
```

附录 6

介绍：Python 编写的基于 ROUGE 评估的 rank 方法

```
import os
import re
import numpy as np
from collections import defaultdict
from rouge import Rouge
import pandas as pd
import openpyxl

script_folder = os.path.dirname(os.path.abspath(__file__))

def read_file(file_path):
    with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
        content = f.read()
    return content

def calculate_rouge_scores(hypothesis, references):
    rouge = Rouge()
    hypothesis = " ".join(hypothesis.split())  # Remove extra whitespaces

    scores_list = []
    for ref in references:
        ref = " ".join(ref.split())  # Remove extra whitespaces
        scores = rouge.get_scores(hypothesis, ref)
        scores_list.append(scores[0])

    if not scores_list:
        return {'rouge-1': {'r': 0, 'p': 0, 'f': 0},
                'rouge-2': {'r': 0, 'p': 0, 'f': 0},
                'rouge-l': {'r': 0, 'p': 0, 'f': 0}}

    avg_scores = {key: {metric: 0 for metric in scores_list[0][key]} for
key in scores_list[0]}
    for score_dict in scores_list:
```

```
        for key in score_dict:
            for metric in score_dict[key]:
                avg_scores[key][metric] += score_dict[key][metric]

    for key in avg_scores:
        for metric in avg_scores[key]:
            avg_scores[key][metric] /= len(scores_list)

    return avg_scores

def calculate_overlap_score(avg_rouge_1_f, avg_rouge_2_f, avg_rouge_l_f):
    rouge_1_weight = 1/3
    rouge_2_weight = 1/3
    rouge_l_weight = 1/3

    overlap_score = (
        rouge_1_weight * avg_rouge_1_f +
        rouge_2_weight * avg_rouge_2_f +
        rouge_l_weight * avg_rouge_l_f
    )

    return overlap_score

# Read files
input_folder = os.path.join(script_folder, "..", "..", "DUC04",
"participating systems result", "2")
summaries = defaultdict(lambda: defaultdict(list))

for file_name in os.listdir(input_folder):
    file_path = os.path.join(input_folder, file_name)
    summary_content = read_file(file_path)

    topic_id, team_id = re.match(r"D(\d+)\.M\.100\.T\.([A-Z\d]+)",
file_name).groups()
    summaries[topic_id][team_id].append(summary_content)

# Calculate scores
team_scores = defaultdict(list)

for topic_id, summaries_by_team in summaries.items():
    reference_summaries = [summary for team_id, summary_list in
summaries_by_team.items() if not team_id.isdigit() for summary in
summary_list]

```

```
    for team_id, team_summaries in summaries_by_team.items():
        if team_id.isdigit():
            topic_scores = []

            for summary in team_summaries:
                rouge_scores = calculate_rouge_scores(summary,
reference_summaries)
                overlap_score = calculate_overlap_score(
                    rouge_scores['rouge-1']['f'],
                    rouge_scores['rouge-2']['f'],
                    rouge_scores['rouge-l']['f']
                )
                topic_scores.append(overlap_score)

            team_scores[team_id].append(np.mean(topic_scores))

# Calculate scores
team_scores = defaultdict(list)

for topic_id, summaries_by_team in summaries.items():
    reference_summaries = [summary for team_id, summary_list in
summaries_by_team.items() if not team_id.isdigit() for summary in
summary_list]

    for team_id, team_summaries in summaries_by_team.items():
        if team_id.isdigit():
            topic_scores = []

            for summary in team_summaries:
                rouge_scores = calculate_rouge_scores(summary,
reference_summaries)
                overlap_score = calculate_overlap_score(
                    rouge_scores['rouge-1']['f'],
                    rouge_scores['rouge-2']['f'],
                    rouge_scores['rouge-l']['f']
                )
                topic_scores.append(overlap_score)

            team_scores[team_id].append(np.mean(topic_scores))

# Calculate average scores for each team
average_team_scores = {team_id: np.mean(scores) for team_id, scores in
team_scores.items()}

```

```
# Print results
for team_id, avg_score in sorted(average_team_scores.items(), key=lambda x: x[1], reverse=True):
    print(f"Team {team_id}: Average Overlap Score: {avg_score:.4f}")

# Create a DataFrame to store the results
results_df = pd.DataFrame(columns=["Team ID", "Average Overlap Score"])

# Add the results to the DataFrame
for team_id, avg_score in sorted(average_team_scores.items(), key=lambda x: x[1], reverse=True):
    results_df = results_df.append({"Team ID": team_id, "Average Overlap Score": avg_score}, ignore_index=True)

# Save the DataFrame to an Excel file
results_df.to_excel(os.path.join(script_folder,"..","Result", "results.xlsx"), index=False)
```

附录 7

介绍：4.6 节代码计算结果

Team ID	Average Overlap Score
65	0.202578227
66	0.200529924
67	0.200460718
35	0.193023004
124	0.191132692
120	0.190772464
19	0.189229009
34	0.18721493
81	0.186961929
102	0.185116473
104	0.184360745
44	0.18370757
121	0.183061736
103	0.180013766
93	0.178168821
45	0.177231383
55	0.175590104
11	0.174782103
56	0.169492637
95	0.169453595
57	0.165364057

140	0.164428118
138	0.16427814
139	0.163535384
94	0.163091112
119	0.161191666
2	0.160455057
117	0.160179115
118	0.157055031
36	0.156958983
123	0.15088628
28	0.144904797
27	0.142321491
29	0.136282674
111	0.104659494