《用 Python 玩转数据》项目——文档相似性比较

相似性比较算法在许多领域有着重要应用。在剽窃检测方面，相似性比较算法可以帮助检查抄袭；在代码管理方面，可以帮助查找大型代码的相似部分，以便进一步优化与修改；在存储方面，可以帮助检测冗余，等等。

借助 Python，我们可以比较简单地实现一些复杂的相似性算法。
相似性比较算法有很多种类，这里介绍的 winnowing 算法基于 2003 年的一篇论文。

一、算法原理和背景知识

算法将文档划分为长度为 n 的连续字符串集合，对每个字符串分片进行哈希，并选择一部分哈希值作为文档的指纹集合。在恰当选择哈希函数的前提下，当两个文档共享一个或多个指纹时，它们很可能共享相同的文档分片。

1. n-gram 模型（n 元模型）

n 元模型是一种概率语言模型，是自然语言处理中一个非常重要的概念，常用于评估字符串之间的差异程度或句子出现的可能性。n-gram 基于马尔科夫假设，即第 n 个词出现的概率只与之前的（n-1）个词有关。

简单举例而言，一串字符“abcdefghijk”，它的 3-gram 表示如下：
abc bcd cde def … ijk
n-gram 的单元可以是字符或词汇。若以词汇为单元，则”… to be or not to be …”的 2-gram 表示如下：
…, “to be”, “be or”, “or not”, “not to”, “to be”, …
在本应用中，对文档使用 n-gram 分割为一组子字符串集合，以方便进一步处理。

2. 哈希算法

哈希（hash）算法是一类算法的统称，根据确定的运算规则对输入进行处理，将较长的串映射为较短的固定长度的值，以达到方便查找、压缩存储空间等目的。使用哈希算法的目的在于使用较短的结果值，可以得知原串的某些性质，同时减少存储和时间上的消耗，所以哈希算法有时又被叫做数据摘要算法。

哈希算法的主要特征是单向性和抗碰撞。两个特征的根本来源都是哈希算法本身将一个大的地址空间映射到一个小空间的原理。单向性是指，无法从哈希值逆向计算出原值，因为对应一个哈希值，可能有很多，甚至无穷多个原值。抗碰撞是指，当我们已知一个原串和它的哈希值，很难构造出另一个串，让它们的哈希值相同。由于哈希函数的值空间一般小于原串的取值空间，对原值作映射运算必定存在碰撞，即多个原串映射到同一个哈希值，一个好的哈希函数应当做到难以通过构造得出碰撞，通俗理解就是分布比较随机，在正常使用时难以出现碰撞，同时在遇到碰撞时有相应的处理方法。

二、文档相似性比较

接下来按照步骤介绍该算法。主要有以下几个步骤：

1. 处理文档
2. 构建分片集合
3. 构建哈希值集合
4. 提取特征指纹
5. 进行比较

1. 文档处理

对于一组想要比较的文档，首先要做的是去除空白符等无用字符，因为这些字符对比较两个文档之间的相似度意义不大，如果不去除，会严重影响比较结果。

示例代码如下：

```
import re

def preprocessing(filename):
    file = open(filename)
    content = file.read()
    space = re.compile(r'\s+')
    content = space.sub('', content)
    content = content.lower()

    return content
```

这里的处理条件可根据个人的需要自行定义。比如，如果处理的是普通文本，可能就不太关心符号，主要关注正文内容，那么可以将标点符号等去除；如果处理的文本与符号密切相关，则可以考虑不去除特殊符号；如果处理的是网络数据，则还需要考虑编码、格式等。

正则表达式是预处理过程的一项重要工具。示例代码中给出了去除空白符的示例。如果想要去除标点符号，只留下正文内容，可以使用如下正则表达式：

```
word = re.compile(r'[^a-zA-Z]')
content = word.sub('', content)
```

关于文档预处理的更多内容，请参考扩展阅读中的链接。

2. 构建分片集合

按照 n-gram 的原理，文档分片就是按照给定的分片大小，将处理后的字符串分割为大小为 n 的字符串集合。

示例代码如下：

```
def generate_n_gram(string, n):
    n_gram = []
    for i in range(len(string)-n+1):
        n_gram.append(string[i:i+n])
    return n_gram
```

3. 构建哈希值集合

接下来的主要工作是对每个文档分片进行运算，计算并存储其哈希值。这样处理后，相似性比较过程可以直接对哈希值进行比较，而不需要进行复杂的字符串比较。

论文中使用的哈希算法 $H(C_1C_2...C_k)=c_1*b^{k-1}+c_2*b^{k-2}*...+c_{k-1}*b+c_k$，其中 b 是确定的基数。

这一算法计算较为简单，同时可以根据 n-gram 的原理简化计算。从 n-gram 的分片过程可以发现，相邻的分片之间有着大部分重叠，后一分片的哈希值可以从前一分片的哈希值经过少量计算推导而出。具体关系如下：

$$H(c_2 \dots c_{k+1}) = (H(c_1 \dots c_k) - c_1 * b^{k-1}) * b + c_{k+1}$$

举例说明：

假设前一分片为 1234，后一分片为 2345，则

$H(1234)=1*b^3+2*b^2+3*b+4$

$H(2345)=2*b^3+3*b^2+4*b+5=(H(123)-1*b^3)*b+5$

可以看出，对大小为 k 的分片，直接计算需要 k-1 次乘法运算（b 的各阶乘方为常数，可以预先算出），而推导式的计算只需要两次乘法运算。当分片大小较大或待处理分片较多时，可以节约大量的乘法运算时间。

示例代码如下：

```
def rolling_hashing(n_gram, Base, n):
    hashlist = []
    hash = 0
    initial = n_gram[0]
    #初始化：Base基数一般设置为素数
    #initial:第一个分片的hash值需要手动计算

    for i in range(n):
        hash += ord(initial[i])*(Base**(n-i-1))
    hashlist.append(hash)

    for i in range(1,len(n_gram)):
        pre = n_gram[i-1]
        present = n_gram[i]

        hash = (hash-ord(pre[0])*(Base**(n-1)))*Base + ord(present[n-1])
        hashlist.append(hash)

    return hashlist
```

4. 提取特征指纹

对于现实应用而言，长文档分片哈希后得到的哈希值集合依然非常大，直接对比是非常低效的，因而需要一些取样算法，帮助我们从整个哈希值集合中取出一部分特征“指纹”，在对比相似性时只对比相同的指纹部分。

一种流行的方法是选择所有整除 p 的哈希值作为指纹，这种方法容易实现，且只保留了原来集合大小的 1/p，但这种方式有一个严重的缺陷，无法限制取得的两个相邻指纹的间距。如果间距过大，很可能造成取样不均匀，忽略了相同的部分；如果设定隔固定间距 W 取一

次指纹，这种非随机化的方法会导致取到的样本均为不重要部分的概率大大增加。因而，需要一种算法，在保证取样间距合理的情况下还要保持一定的随机性。
Winnowing 算法的基本思路是设定一个大小为 W 的滑动窗口，当窗口从前往后沿哈希值数组滑动时，将每个窗口中最小的哈希值保留下来，如果有多个最小值，选择最右边出现的作为指纹。

> DEFINITION 1 (WINNOWING). *In each window select the minimum hash value. If there is more than one hash with the minimum value, select the rightmost occurrence. Now save all selected hashes as the fingerprints of the document.*

算法的思路是：确定一个阈值 t，保证任何长度超过 t 的匹配均会被检测到；确定一个阈值 n，将所有长度小于 n 的匹配视为噪音忽略。n 的限制已经由前面的 n-gram 分片算法完成，接下来需要确定如何保证 t 这一条件。显然，只要保证至少每(t-n+1)个连续的哈希之中有一个特定值被选取出来，就可以保证检测所有长度超过 t 的特征被检测到。本算法选择 W=t-n+1 的窗口大小，并选择每个窗口的最小值作为特定值。
严格证明，请参考扩展阅读中的原始论文。
参考代码如下：

```
def winnowing(hashlist, t, n):
    window = t-n+1
    minValue = minPos = 0
    fingerprint = {}
    for i in range(len(hashlist)-window+1):
        temp = hashlist[i:i+window]
        minValue = temp[0]
        minPos = 0
        for j in range(window):
            if temp[j]<=minValue:
                minValue = temp[j]
                minPos = j
        if (i+minPos) not in fingerprint.keys():
            fingerprint[i+minPos] = minValue
    return fingerprint
```

（注：此处的返回结果为 Python 字典类型，包括取得的哈希值以及该值在字符串中的位置，如果不需要比对相似部分所在的位置，只比较总体的相似度，则可以不返回位置信息）

5. 进行比较

比较操作即检查两篇文档的指纹是否存在相同值，参考代码如下：

```
def comparison(fingerprint_1, fingerprint_2):
    count = 0
    size = min(len(fingerprint_1),len(fingerprint_2))
    for i in fingerprint_1.values():
        for j in fingerprint_2.values():
            if (i==j):
                count += 1
                break
    return count/size
```

示例代码可在编辑器中查看，也可直接点击运行，只要源代码路径下存在 test_1.txt 和 test_2.txt，即可比较并输出结果。

三、扩展阅读

1. Winnowing 算法的原始论文：Winnowing: local algorithms for document fingerprinting
2. 关于 Winnowing 算法的原理讲述，可以参考基于 K-gram 的 winnowing 特征提取剽窃查重检测技术（概念篇）
3. 关于文本挖掘的流程，主要是文档的预处理，可以参考用 Python 做文本挖掘的流程
4. 关于 Python 正则表达式的相关说明，可以参考 Python 正则表达式指南

四、参考代码

```
import re
import os

def preprocessing(filename):
    file = open(filename)
    content = file.read()
    word = re.compile(r'[^a-zA-Z]')
    content = word.sub('', content)
    content = content.lower()

    return content

def generate_n_gram(content, n):
    n_gram = []
    for i in range(len(content)-n+1):
        n_gram.append(content[i:i+n])
    return n_gram

def rolling_hashing(n_gram, Base, n):
    hashlist = []
    hash = 0
    initial = n_gram[0]
    #初始化：Base 基数一般设置为素数
    #initial:第一个分片的 hash 值需要手动计算

    for i in range(n):
        hash += ord(initial[i])*(Base**(n-i-1))
    hashlist.append(hash)
```

```
    for i in range(1,len(n_gram)):
        pre = n_gram[i-1]
        present = n_gram[i]

        hash = (hash-ord(pre[0])*(Base**(n-1)))*Base + ord(present[n-1])
        hashlist.append(hash)

    return hashlist

def winnowing(hashlist, t, n):
    window = t-n+1
    minValue = minPos = 0
    fingerprint = {}
    for i in range(len(hashlist)-window+1):
        temp = hashlist[i:i+window]
        minValue = temp[0]
        minPos = 0
        for j in range(window):
            if temp[j]<=minValue:
                minValue = temp[j]
                minPos = j
        if (i+minPos) not in fingerprint.keys():
            fingerprint[i+minPos] = minValue
    return fingerprint

def comparison(fingerprint_1, fingerprint_2):
    count = 0
    size = min(len(fingerprint_1),len(fingerprint_2))
    for i in fingerprint_1.values():
        for j in fingerprint_2.values():
            if (i==j):
                count += 1
                break
    return count/size

if __name__ == '__main__':
    print('分片大小为 5')
    print('检测阈值为 9')
    dirpath = os.getcwd()
    path_1 = dirpath + "\\test_1.txt"
    path_2 = dirpath + "\\test_2.txt"
    fingerprint_1                                                                    =
winnowing(rolling_hashing(generate_n_gram(preprocessing(path_1), 5),17, 5),9,5)
```

```
    fingerprint_2 =
winnowing(rolling_hashing(generate_n_gram(preprocessing(path_2), 5),17, 5),9,5)
    print("相似度：")
    print(comparison(fingerprint_1,fingerprint_2))
    input()
```

五、测试文本

test_1.txt

In many applications it is useful to record not only the fingerprints of a document, but also the position of the fingerprints in the document. For example, we need positional information to show the matching substrings in a user interface. An efficient implementation of winnowing also needs to retain the position of the most recently selected fingerprint. Figure 2(f) shows the set of [fingerprint, position] pairs for this example (the first position is numbered 0). To avoid the notational complexity of indexing all hashes with their position in the global sequence of hashes of k-grams of a document, we suppress most explicit references to the position of k-grams in documents in our presentation.

test_2.txt

An efficient implementation of winnowing needs to keep the pos of most recently selected fingerprint, too. To avoid the complexity of indexing hashes with position in global sequence of hashes of k-grams in a document. It is very useful in many applications to not only record the fingerprints of documents, but also to record the fingerprints. For example, if we need positions to show the matching substrings in a user's interface.

创建文本文件将两段文本内容分别存入。

六、其他方式

相似性计算也可以利用著名的 Python 自然语言处理库 Gensim 模块（或 SnowNLP 等），它可以很方便地将文档词袋化，并将其基于 tf-idf 等模型转化成向量模式，训练主题数为 N 的模型（例如 LSI 模型），并用待检索的文档向量初始化一个相似度计算的对象，再通过 LSI 模型将某一行映射到 N 个 topic 模型空间上，与其他行计算相似度。代码形如：

```
…
corpus = [dictionary.doc2bow(line) for line in filtered_text]
tfidf_text = models.TfidfModel(corpus)
```

```
corpus_tfidf = tfidf_text[corpus]
lsi = models.LsiModel(corpus_tfidf, id2word=dictionary, num_topics=N)
index = similarities.MatrixSimilarity(lsi[corpus])
…
```

更详细的内容可参考 https://radimrehurek.com/gensim/tutorial.html。