Document Distance

小组成员：饶远(S211000801) 高瑛璇(S2110W0943) 黄姚昊(S211000793) 姚成伟(S211000859)

一、问题介绍

文档距离就是给定两份文件，编写一个程序来计算它们有多相似，并且算法要求很快的速度。Document Distance 广泛应用于搜索引擎等领域。

二、问题分析

1.“单词”指的是字母和数字(alphanumeric)。

2.每个文档统计完词频后得到的 list，可看作一个向量。

3.两个文档间的相似度，是相似的单词除以总的单词，类似于两个向量的夹角公式。

所以程序的算法可以描述为 3 步：

1.将每个文档分离为单词。

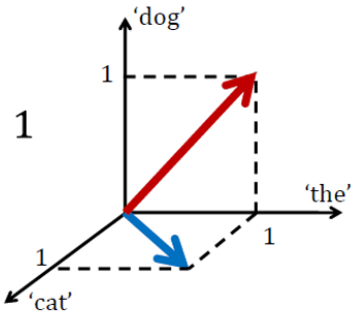
2.统计词频。

3.计算点积（并做除法）：

$$\theta(D_1, D_2) = \arccos \frac{D_1 \cdot D_2}{||D_1|| \cdot ||D_2||}$$

- Example:
  - D_1 = “the cat”
  - D_2 = “the dog”

$D_1 \cdot D_2 = 1$



- Similarity between vectors?
  - **Dot product:**

$$D_1 \cdot D_2 = \sum_w D_1(w) \cdot D_2(w)$$

三、代码及优化过程

1、docdist1.py

docdist1.py 是我们第一个程序，是一个很直接的解决方案，这也是我们优化的基础，docdist{2-8}.py 显示了改进运行时间的算法优化。

我们从理解直接实现的结构开始，然后我们将查看每个后续版本中的更改。

我们首先看一下 main，从更高的层次了解程序中正在发生的事情：

```
docdist1.py > main
1   def main():
2       if len(sys.argv) != 3:
3           print("Usage: docdist1.py filename_1 filename_2")
4       else: filename_1 = sys.argv[1]
5       filename_2 = sys.argv[2]
6       sorted_word_list_1 = word_frequencies_for_file(filename_1)
7       sorted_word_list_2 = word_frequencies_for_file(filename_2)
8       distance = vector_angle(sorted_word_list_1,sorted_word_list_2)
9       print("The distance between the documents is: %0.6f (radians)"%distance)
```

该 main 函数处理命令行参数，为每个文档调用 word_frequencies_for_file 函数，然后调用 vector_angle。这些方法如何与概述的三个操作相匹配？word_frequencies_for_file 函数负责操作 1（将每个文档拆分为单词）和操作 2（计算单词频率），然后 vector_angle 函数负责操作 3（计算点积）。

然后是 word_frequencies_for_file 函数：

```
11  def word_frequencies_for_file(filename):
12      line_list = read_file(filename)
13      word_list = get_words_from_line_list(line_list)
14      freq_mapping = count_frequency(word_list)
15      return freq_mapping
```

该方法首先调用 read_file，它返回输入文件中的行列表。我们将省略 read_file 代码，因为它不是特别有用，并且我们将假设读取文件的运行时间与输入文件的大小成比例。get_words_from_line_list 函数用于从行列表中获取单词，计算操作 1（将每个文档拆分为单词）。之后，count_frequency 将单词列

表转换为文档向量（操作 2）。

接下来是 get_words_from_line_list 函数

```
17  def get_words_from_line_list(L):
18      word_list = []
19      for line in L:
20          words_in_line = get_words_from_string(line)
21          word_list = word_list + words_in_line
22      return word_list
23
24  def get_words_from_string(line):
25      word_list = []
26      character_list = []
27      for c in line:
28          if c.isalnum():
29              character_list.append(c)
30          elif len(character_list)>0:
31              word = "".join(character_list)
32              word = word.lower()
33              word_list.append(word)
34              character_list = []
35      if len(character_list)>0:
36          word = "".join(character_list)
37          word = word.lower()
38          word_list.append(word)
39      return word_list
```

get_words_from_string 函数从字符串获取单词在输入文件中取一行，并将其拆分为单词列表。它是用的逐行分析。运行时间为 O（k），K 是行的长度。

get_words_from_line_list 调用 get_words_from_string 函数，并将列表合并到一个大列表中。第 21 行看起来很无辜，但却是一个巨大的性能杀手，因为使用+对 $\frac{W}{k}$ 个长度为 k 的列表组合是 $O(\frac{W^2}{k})$的。

get_words_from_line_list 的输出是一个单词列表，如['a', 'cat', 'in', 'a', 'bag']，word frequencies from file 将输出传给 count frequency，它将其转换为一个文档向量，该向量统计每个单词的出现次数，如[['a', 2], ['cat', 1], ['in', 1], ['bag', 1]]。

```
41  def count_frequency(word_list):
42      L = []
43      for new_word in word_list:
44          for entry in L:
45              if new_word == entry[0]:
46                  entry[1] = entry[1] + 1
47                  break
48              else:
49                  L.append([new_word,1])
50      return L
```

上面的实现通过获取输入列表中的每个单词并在表示正在构建的文档向量的列表中查找来构建文档向量。对于包含所有不同单词的文档，在最坏的情况下，这需要 O（l*W^2）时间，其中 W 是文档中的单词数，l 是平均单词长度。count frequency 是文 word frequencies for file 中的最后一个函数调用。

下一步是 vector angle，实现步骤 3：

```
52  def vector_angle(L1,L2):
53      numerator = inner_product(L1,L2)
54      denominator = math.sqrt(inner_product(L1,L1)*inner_product(L2,L2))
55      return math.acos(numerator/denominator)
```

该方法是距离度量的简单实现：

$$\arccos\left(\frac{L1 \cdot L2}{|L1||L2|}\right) = \arccos\left(\frac{L1 \cdot L2}{\sqrt{(L1 \cdot L1)(L2 \cdot L2)}}\right)$$

inner_product 如下：处理运算。

```
57  def inner_product(L1,L2):
58      sum = 0.0
59      for word1, count1 in L1:
60          for word2, count2 in L2:
61              if word1 == word2:
62                  sum += count1 * count2
63      return sum
```

这是一个简单的内积实现，它根据整个第二个列表检查第一个列表中的每个单词。第 3 行和第 4 行的嵌套循环给出了算法的运行时间 Θ（L1*L2），其中 L1 和 L2 是文档向量的长度（每个文档中唯一的单词数）。

分析可知 docdist1.py 的性能如下：

Method	Time
get_words_from_line_list	$O(\frac{W^2}{k}) = O(W^2)$
count_frequency	$O(WL)$
word_frequencies_for_file	$O(W^2)$
inner_product	$O(L_1L_2)$
vector_angle	$O(L_1L_2 + L_1^2 + L_2^2) = O(L_1^2 + L_2^2)$
main	$O(W_1^2 + W_2^2)$

2、docdist2.py

docdist1 的 profiler 输出显示，最大的时间消耗是 get words from line list。问题是，当使用+运算符连接列表时，它需要创建一个新列表并复制其两个操作数的元素。将+替换为 extend 可使运行时提高 30%。

```
17  def get_words_from_line_list(L):
18      word_list = []
19      for line in L:
20          words_in_line = get_words_from_string(line)
21          word_list.extend(words_in_line)
22      return word_list
```

extend 将元素列表中的所有元素添加到元素列表中，以 Θ（1+m）表示，而+需要创建一个新列表，因此需要 Θ（1+n+m）时间，因此，将 $\frac{W}{k}$ 个长度为 k 的列表组合是 $\sum^{\frac{W}{k}} k = \Theta(W)$ 的。

docdist2.py 的性能如下：

Method	Time
get_words_from_line_list	$O(W)$
count_frequency	$O(WL)$
word_frequencies_for_file	$O(WL)$
inner_product	$O(L_1L_2)$
vector_angle	$O(L_1^2 + L_2^2)$
main	$O(W_1L_1 + W_2L_2)$

3、docdist3.py

分析 docdist2 后，将 count frequency 和 inner product 作为优化的良好目标。我们会通过切换到快速算法生成，加快 inner product。但是，该算法假定文档向量中的单词已排序。例如[['a', 2], ['cat', 1], ['in', 1], ['bag', 1]] 需要变成[['a', 2], ['bag', 1], ['cat', 1], ['in', 1]]。

所以我们增加了一步排序：

```
def word_frequencies_for_file(filename):
    line_list = read_file(filename)
    word_list = get_words_from_line_list(line_list)
    freq_mapping = count_frequency(word_list)
    insertion_sort(freq_mapping)
    return freq_mapping
```

这里使用的是插入排序。

最后发现 docdist3.py 的性能如下：

Method	Time
get_words_from_line_list	$O(W)$
count_frequency	$O(WL)$
insertion_sort	$O(L^2)$
word_frequencies_for_file	$O(WL + L^2) = O(WL)$
inner_product	$O(L_1 + L_2)$
vector_angle	$O(L_1 + L_2)$
main	$O(W_1L_1 + W_2L_2)$

4、docdist4.py

分析 docdist3 后，发现 count frequency 是消耗最多时间的。

```
def count_frequency(word_list):
  D = {}
  for new_word in word_list:
    if new_word in D:
      D[new_word] = D[new_word]+1
    else:
      D[new_word] = 1
  return D.items()
```

新的实现使用 Python 字典。字典是使用哈希表实现的。哈希表的显著特点

是，使用 dictionary[key]=value 插入元素和使用 dictionary[key]查找元素都在 O(1)的时间运行。

新的实现没有将正在构建的文档向量存储在列表中，而是使用字典。键是文档中的单词，值是每个单词在文档中出现的次数。因为插入（第 5 行）和查找（第 7 行）都需要（1）个时间，所以用 W 个单词构建文档向量需要 O（W）个时间。

最后发现 docdist4.py 的性能如下：

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(W)$
`insertion_sort`	$O(L^2)$
`word_frequencies_for_file`	$O(W + L^2) = O(L^2)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(L_1^2 + L_2^2)$

5、docdist5.py

分析 docdist4 后，发现可以做出以下修改：

```
translation_table = string.maketrans(string.punctuation+string.uppercase,
                    " "*len(string.punctuation)+string.lowercase)

def get_words_from_string(line):
  line = line.translate(translation_table)
  word_list = line.split()
  return word_list
```

这个迭代简化了 get words from string 将行拆分为单词。首先，标准库函数 string.translate 用于将大写字符转换为小写，以及将标点符号转换为空格。其次 split 使用字符串拆分方法将行拆分为单词。

此更改的主要好处是 23 行代码被 5 行代码替换。这使得实现更容易分析。另一个好处是 Python 标准库中的许多函数都是直接用 C 实现的，这使它们具

有更好的性能。尽管运行时间是渐近相同的，但 C 代码的隐藏常量比 docdist1 中的 Python 代码要好得多。

6、docdist6.py

分析 docdist5 后，发现上面用的插入排序比较慢，改成归并后如下：

```
def word_frequencies_for_file(filename):
  line_list = read_file(filename)
  word_list = get_words_from_line_list(line_list)
  freq_mapping = count_frequency(word_list)
  freq_mapping = merge_sort(freq_mapping)
  return freq_mapping
```

性能如下：

Method	Time
get_words_from_line_list	$O(W)$
count_frequency	$O(W)$
merge_sort	$O(L \log L)$
word_frequencies_for_file	$O(W + L \log L) = O(L \log L)$
inner_product	$O(L_1 + L_2)$
vector_angle	$O(L_1 + L_2)$
main	$O(L_1 \log L_1 + L_2 \log L_2)$

7、docdist7.py

分析 docdist6 后，发现切换到合并排序大大缩短了运行时间。但是，如果我们查看 docdist6 的分析器输出，就会发现 merge 是运行时占用空间最大的函数。合并排序在实践中的性能是非常好的，因此似乎使我们的代码更快的唯一方法是完全取消排序。

此迭代不再使用文档向量的排序列表表示，而是使用 docdist4 中引入的 Python 字典表示。count frequency 已经在内部使用了该表示，因此我们只需要删除将 Python 字典转换为列表的代码。

```
def count_frequency(word_list):
  D = {}
  for new_word in word_list:
    if new_word in D:
      D[new_word] = D[new_word]+1
    else:
      D[new_word] = 1
  return D
```

这种方法仍然需要 O（W）时间来处理 W 个单词的文档。

word frequencies for file 也不用排序了，改用 count frequency 的字典。

```
def word_frequencies_for_file(filename):
    line_list = read_file(filename)
    word_list = get_words_from_line_list(line_list)
    freq_mapping = count_frequency(word_list)
    return freq_mapping
```

接下来，inner_product 也使用字典查找而不是合并排序列表。

```
def inner_product(D1,D2):
  sum = 0.0
  for key in D1:
    if key in D2:
      sum += D1[key] * D2[key]
  return sum
```

该逻辑与 docdist1 中简单的内积非常相似。在第二个文档向量中查找第一个文档向量中的每个单词。但是，因为文档向量是字典，所以每个都需要 O(1) 时间，内积以 O（L1）个时间运行，其中 L1 是第一个文档向量的长度。

性能如下：

Method	Time
get_words_from_line_list	$O(W)$
count_frequency	$O(W)$
word_frequencies_for_file	$O(W)$
inner_product	$O(L_1 + L_2)$
vector_angle	$O(L_1 + L_2)$
main	$O(W_1 + W_2)$

8、docdist8.py

分析 docdist7 后，此时，我们解中的所有算法都是渐近最优的。我们可以很容易地证明这一点，注意到 3 个主要操作中的每一个都按照其输入大小的时间比例运行，并且每个操作都需要读取其所有输入以产生其输出。但是，仍然有优化和简化代码的空间。

没有理由逐行阅读每个文档，然后将每一行分解成文字。最后一次迭代过程将每个文档读入一个大字符串，并立即将整个文档分解为单词。

修改如下:

```
def get_words_from_text(text):
    text = text.translate(translation_table)
    word_list = text.split()
    return word_list

def word_frequencies_for_file(filename):
    text = read_file(filename)
    word_list = get_words_from_text(text)
    freq_mapping = count_frequency(word_list)
    return freq_mapping
```

性能如下:

Method	Time
get_words_from_text	$O(W)$
count_frequency	$O(W)$
word_frequencies_for_file	$O(W)$
inner_product	$O(L_1 + L_2)$
vector_angle	$O(L_1 + L_2)$
main	$O(W_1 + W_2)$

四、结果展示

docdist1	initial version	
docdist2	add profiling	192.5 sec
docdist3	replace + with extend	126.5 sec
docdist4	count frequencies using dictionary	73.4 sec
docdist5	split words with string.translate	18.1 sec
docdist6	change insertion sort to merge sort	11.5 sec
docdist7	no sorting, dot product with dictionary	1.8 sec
docdist8	split words on whole document, not line by line	0.2 sec