英文单词拼写错误自动检查系统

包利强，雷雪林，戴君义

2017 年 7 月 15 日

目录

1 课题简介 **2**

2 项目目标 **2**

3 国内外相关工作 **2**

3.0.1 拼写纠错相关数据结构 2

3.0.2 独立单词拼写纠错 3

4 系统主要模块流程 **4**

5 核心思想和算法描述 **5**

5.1 检错系统 5

5.2 纠错系统 5

5.3 一些细节 8

6 代码运行简述 **9**

7 系统评测 **10**

7.1 测试数据 10

7.2 评测标准 10

7.3 评测结果 11

8 组员分工 **11**

8.1 包利强 11

1 课题简介

在英语写作中，单词的拼写错误是一种较为常见的错误，因此英文单词自动纠错的研究成为了自然语言处理技术的重要子课题。课题 11 要求我们实现一个英文文本中单词拼写错误自动检查系统，如果在检查的同时还要求进行纠错错，那么这个系统将包括两个基本的部分，即检错（detecting）和纠错（correcting）。在现代自然语言处理的研究中，英文单词拼写纠错系统又可以按照不同的情境分别进行研究，其一是上下文无关单词的检错和改错（*context-free word error detection and correction or isolated-word error detection and correction*），在该情境下，每个单词的检查都是与它的上下文环境相独立的，为了检测该单词是否正确，很自然的一个做法是在词典中搜索该单词，如果搜索成功则拼写正确，否则错误，在这种方法中，算法对词典非常敏感。然而这个方法的一个很大的缺点是无法对那些拼写正确但是不符合上下文的单词进行检错，因此依赖上下文的检错和改错（*context-dependent error detection and correction*）也是一个非常重要的研究课题，但由于这个课题需要语法相关的分析，因此比较复杂 [1]。

2 项目目标

目前在自然语言处理研究中，拼写检查和纠错方面的研究已经进行了很长一段时间，有了很多成果和基础，也有了不少开源的拼写检查与纠错的成熟系统，主要以 GNU Aspell 和 Hunspell 为代表。我们小组在阅读相关文献和实践的基础上，将动手实现一个简单的类 GNU Aspell[11] 系统，由于在 1中所解释的原因，我们将只实现上个一下文无关的单词检错和改错系统。拼写错误改错意味着要对文本中出现的错误单词提供可能的正确单词建议，一般是一个概率递减的建议列表，但在自动改错时，我们只选择概率最大的一个单词替换错拼单词即可。为了方便表述，我们将这个简单的拼写纠错器称为 Bspell。

3 国内外相关工作

3.0.1 拼写纠错相关数据结构

如 1中所说，检测一个单词是否拼写错误看起来非常简单，我们只需要在一个词典中对该单词进行查询即可，但这个方法同时也有很大的问题，首先一个包含所有正确单词的词典文件可能会非常大，查询会造成非常糟糕的空间和时间复杂度；其次，错误的拼写在很多时候会产生可以在词典中查询到的正确单词，这种情况叫做**真词错误**；最后，过大的词典文件会包含很多怪异的单词，这会增加产生真词错误的可能。

20 世纪 80 年代末到 90 年代初，有研究者发现，对于不同的语言，所需要的适当的词典大小是不同的，对于屈折度不高的语言如英语来说，包含 50000-200000 单词的词典的效果是较好的 [2, 3, 4]。而对于其他一些屈折度较高的语言，所需要词典的大小可能会非常大。

在经典的数据结构中，可以满足词典的快速查询的一个实现是哈希表 [5]。但传统的哈希表具有的一个缺点是哈希函数的选择和合适的表规模设置，否则会产生严重的碰撞问题。1997 年，Czech 等人 [6] 所提出的最小完美哈希（minimal perfect hashing）解决了碰撞的问题，但同时也要求一个非常大的基本上和词典规模相当的哈希表。

另外一个经常用来存储词典的数据结构是前缀树（Trie）[5]。这是一种有序树，用于保存关联数组，其中的键通常是字符串，与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。前缀树能够有效加速词典的查询，但它的大小也是依赖于词典的大小，为了减小前缀

树的规模，很多研究者也做出了自己的努力，如著名的 C-trie[7]，PATRICIA[8]，以及 Bonsai[9]。

也有研究者使用一种无环确定性有限自动机（Acyclic Deterministic Finite Automaton，ADFA）[1]，可以看做是前缀树的扩展。如果一颗完全前缀树的等价子树已经全部合并，那么我们就得到了一颗极小 ADFA，其基本思想与前缀树类似。

还有一种方法是直接保存每个单词的词根和相应的生成规则，这样我们就知道了怎么生成每个单词的正确形式。这种方法广泛使用在开源的拼写检查器如 Ispell[10] 和 Aspell[11] 中。如在 Ispell 中的一些例子：boolean/S，这意味着这个单词的复数形式 booleans 也是正确的；frizzle/DGS，则意味着 frizzle,frizzled,frizzles, 以及 frizzling 都是正确的。

3.0.2 独立单词拼写纠错

独立单词的纠错意味着要为**非词错误**提供一个或多个正确的单词建议，如果有多个建议，则它们应该按照可能性大小依次排序。如对于一个非词错误 *speling*，*spelling* 比 *spilling* 可能性更大，但是也不能完全确定。

一种最简单的方法是基于这样一个假设：人们在打字时通常只会犯几种错误，因此对每一个错误单词，我们需要弄明白使用一个正确单词来产生它的最小的基本操作数（如插入、删除和替换），所需的步数越小，则该正确单词产生它的概率越大。这个方法叫做最小编辑距离（minimal edit distance）方法 [12]，Damerau 提出后又由 Wagner 作了进一步研究 [13]。1966 年，一种建立在插入、删除和移位操作上的相似方法被提出 [14]。但是这种方法只能解决三大拼写错误中的一种，即**误敲**，而不能同时解决其他错误。一种直接的实现方法是对词典中的每一个单词都和非词错误进行对比，这种作法比较耗时；一种更快的做法 [15] 可以使得词典中要进行比较的单词数量减少一个数量级，如果非词中的错误个数较小的话。

在**相似性键技术（similarity key technique）**中，词典中的每个单词都被赋予一个键，在和非词的比较中，只有键才会进行比较，那些和非词的线索非常相似的键对应的单词会被选为建议单词。由于只有相似的键才会参与比较，所以这个方法更加有效。这个方法首先是基于 SOUNDEX 系统 [16] 提出的，这个系统是用来对按发音转译的名字进行纠错。后来，一个相似的但是更加准确的方法被开发出来，称为 SPEEDCOP[17]。这个系统提供了两种线索的实现，即概要（skeleton）和省略（omission）。但这种方法只适应于误敲的情况。

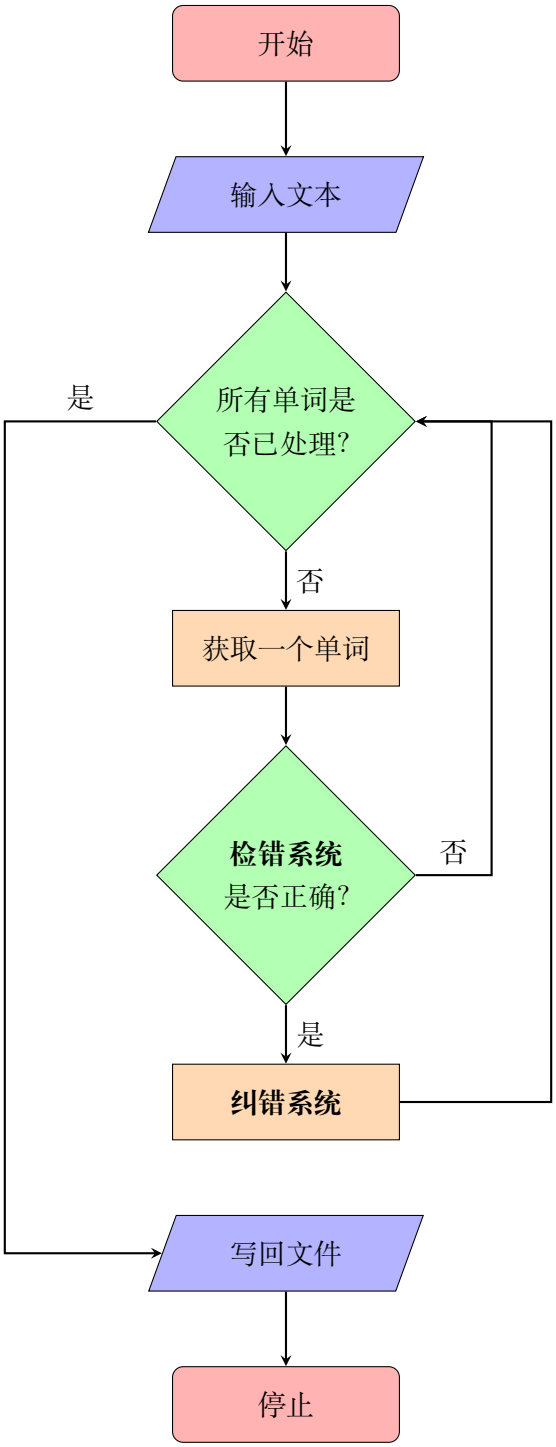
基于规则（Rule-based）的方法适用于更加普遍的错误类型，这种方法考虑的是非词向词典中正确单词的转变。第一个基于知识的系统提出于 1983 年 [18, 19]，所依赖的规则是从 1000 个拼写错误中得到的。这种方法有两个问题，首先是知识或规则必须通过对真实拼写错误进行实验才能得到，并且必须融合进算法之中，其次是对所获得的单词列表进行验证，判断它们是否属于词典非常困难，这使得整个过程非常耗时。但是这种方法具有的一个很好的优点是它对所有三种拼写错误都能进行处理。

在基于概率的方法（Probabilistic techniques）中，不同的编辑操作产生错误的概率是不同的。对字母替代、字母插入、字母删除和字母移位这些基本操作的的概率研究构成的概率方法的基础。在 Church 和 Gale 提出的算法 [20] 中，这些单词通过分析语料库中的大量的单词获得，建议单词是按照生成非词的概率大小来进行排序的。

语音相似性（phonetic similarity）方法适用于对拼写错误的单词进行纠错。这种方法有一个假设，即是打字者确切地知道所要打的单词的发音，但是不知道正确的拼写方式，这样的人常常使用不正确的字形来表达音素。语音相似性可以有很多种表达方式，如 SOUNDEX 编码 [16] 即是一种，其他的如 PHONIX[21]，MetaPhone[22]，以及 Double MetaPhone[23]。

4 系统主要模块流程

通过对相关文章的调研，我们组提出的自动纠错系统主要分为两个模块，即**检错系统**和**纠错系统**。检错系统所使用的算法为**哈希表匹配法**，而纠错所使用的算法主要是**基于最长公共子串的 Needleman-Wunsch 算法**。主要的纠错流程为：



5 核心思想和算法描述

5.1 检错系统

由 3.0.1的描述我们知道，进行单词检错最简单的一种方法是**哈希表查询法**，为了实现这种技术，我们需要建立一个规模与词典大小相当的哈希表词典对象，然后将等检测的单词进行哈希匹配，如果匹配成功，则意味着待检测的单词在词形上没有错误，否则该单词拼写错误。在具体的实现过程中，我们首先使用了一个大小为 69903 的英文单词表，这个单词表已经足以包含我们在日常生活中所遇到的单词，但是在那些专有名词出现较多的文章中，我们建议使用专用的单词表进行代替；其次，我们使用 python 内置的 dict 数据结构来构造哈希表，因为 dict 的内部实现就是基于哈希函数的，这样可以在最大程度上简化我们的工作，同时还可以收到良好的效果，但有一点儿什得注意，python 中的哈希函数并非前面提到的最小完美哈希函数，但对于一个具有 6 万多大小的列表来说，这个碰撞度是完全可以忽略的。

主要的代码如下：

```
def init_dict(self, voc_file):
    """
    initialize class owned dict with vocabulary file
    :param voc_file: vocabulary file
    :return:
    """
    f = open(voc_file, 'r')
    for line in f:
        tmp_line = line.strip()
        if not self.dict[tmp_line]:
            self.dict[tmp_line] = True

def search(self, word):
    """
    search the word in the dict
    :param word: word to search
    :return: whether the word is in the dict
    """
    is_word = self.dict[word.strip()]
    if not is_word:
        self.dict.pop(word)
    return is_word
```

5.2 纠错系统

我们在调研的过程中发现，基于**最小编辑距离**的算法在独立单词纠错的应用中得到了广泛的应用，如非常经典的 **LG 算法**，但与此相似的**基于最长公共子串**的算法却鲜见踪迹，因此我们决定使用基于最长公共子串的 **Needleman-Wunsch 算法**来完成对单词的检错工作。具体的算法描述为：

- 形式化定义。
  给定两个单词 S 和 T，S 的长度为 m，T 的长度为 n，然后定义 Alignment 表示产生式，即单词 S 变成 T 的过程记录。我们运用最长公共子串的概念对 Alignment 进行打分，分数越高，则代表两个单词的相似度越高。打分原则为：

$$d(S,T) = \sum_{i=1}^{min(m,n)} \delta(S[i],T[i])$$

其中，关于 $\delta(S[i],T[i])$ 我们有三种情况：如果 S 的第 i 个字符与 T 的第 i 个字符一致，则 $\delta(S[i],T[i])=1$（表示得到一分）；如果不一致，则 $\delta(S[i],T[i])=-1$；如果少敲（如 S 的第 i 个字符多了出来），则 $\delta(S[i],T[i])=-3$。

举一个简单的例子为：

$$S:OC-URRA-NCE$$
$$T:OCCU-PATION-$$

$$\text{Alignment}:d(S,T)=1+1-3+1-3-3-1+1-3-3-3+1-3-3=-20$$

Alignment 十分重要，它不仅可以识别词典中的相似度高的单词，还可以记录两个单词之间出现错误的具体信息。

- 动态规划求解。
  形式化定义以后，我们求两个英文单词的相似度的问题就转换成了有两个字符串 S 和 T，我们应该如何对其进行加空格使得我们的得分最高的问题。我们将给出的字符串分成更小的字符串来解决会更加方便。首先考虑最后的一个字母，假设 S 单词的最后一个字母为 a，这个 a 通过 T 变换有三种情况：

  1. 如果 T 的最后一个单词也是 a，则他们进行匹配了；
  2. 如果 a 是我们多敲的，则 T 中的 a 需要我们进行插入操作变换而来；
  3. 如果 a 是我们少敲的，则 T 中的 a 需要我们进行删除操作变换而来。

  根据这三种情况，我们得到如下对应：

  1. 如果 S[m] 与 T[n] 的最后一个单词形成匹配，则我们子问题就变成了 S[1,...,m-1] 与 T[1,...,n-1] 的对齐问题；
  2. 如果 S[m] 与空格匹配，则我们子问题就变成了 S[1,...,m-1] 与 T[1,...,n] 的对齐问题；
  3. 如果 T[m] 与空格匹配，则我们子问题就变成了 S[1,...,m] 与 T[1,...,n-1] 的对齐问题。

  我们将问题的最优解记为 $\text{OPT}(i,j)=S[1\cdots i]alignmentT[1\cdots j]$，我们可以得到下面的最优化子结构：

$$\text{OPT}(i,j)=max\begin{cases}\delta(S_i,T_j)+\text{OPT}(i-1,j-1),\\ \delta('-',T_j)+\text{OPT}(i,j-1),\\ \delta(S_i,'-')+\text{OPT}(i-1,j),.\end{cases}$$

  即枚举当前单元的三种可能，在其中取最大值就可以了。

  我们可以通过具体表格来说明打分过程：
  Alignment : $d(\text{"}OCURRANCE\text{"},\text{"}OCCURRENCE\text{"})$

	″	O	C	U	R	R	A	N	C	E
″	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−14	−17
U	−12	−8	−4	0	0	−3	−6	−9	−12	−15
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−1	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−7
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

中间的单元都是由相邻的三个单元变换而来，我们的最后得分为 4。这个得分本身有三个来源（即它的上方，左方，和左上方），我们可以通过不断回溯的方法找到其对应的 alignment，使得 S 变成 T。为了得到更加准确的评分，我们可以回溯多次，然后取平均，这样可以得到一个比较常见的模式。

Bspell 系统主要的动态规划代码为：

```
def match(self, s, t):
    """
    match two strings and return the final score
    :param s: string 1
    :param t: string 2
    :return: score
    """

    self.__init__(s, t)

    m = len(s) + 1
    n = len(t) + 1

    for i in range(m):
        self.opt[i, 0] = -3 * i
    for j in range(n):
        self.opt[0, j] = -3 * j

    for i in range(1, m):
        for j in range(1, n):
            if self.s[i - 1] == self.t[j - 1]:
                d = 1
            else:
                d = -1

            self.opt[i, j] = max(
                self.opt[i - 1, j - 1] + d,
                self.opt[i - 1, j] - 3,
                self.opt[i, j - 1] - 3)

    return self.opt[m - 1, n - 1]
```

用来对最长公共子串进行回溯的代码为：

```
def backtrack(self):
    """
    find the most possible case
    :return: the most possible case
    """

    i, j = len(self.s), len(self.t)
    a = b = ''

    while i != 0 or j != 0:
        if i == 0:
            j -= 1
            a += '-'
            b += self.t[j]
        elif j == 0:
            i -= 1
            a += self.s[i]
            b += '-'
        else:
            x, y, z = self.opt[i - 1, j - 1], self.opt[i - 1, j], self.opt[i, j - 1]
            if x >= y and x >= z:
                i -= 1
                j -= 1
                ts = self.s[i]
                tt = self.t[j]
                if ts == tt:
                    a += ts
                    b += tt
                else:
                    a += '*'
                    b += '*'
            elif y > z:
                i -= 1
                a += self.s[i]
                b += '-'
            elif y < z:
                j -= 1
                a += '-'
                b += self.t[j]
            else:
                if i >= j:
                    i -= 1
                    a += self.s[i]
                    b += '-'
                else:
                    j -= 1
                    a += '-'
                    b += self.s[j]

    return a[::-1], b[::-1]
```

5.3 一些细节

- 建议序列二次排序。
  经过 Needleman-Wunsch 算法对词典中的与错拼单词长度接近的单词进行动态规划匹配后，会产生一个排序列表，这个列表中的前几名可能与错拼单词都拥有相同长度的最长公共子串，

如果直接将这个列表进行返回，效果会比较差，比如有的单词错拼字母位置靠前，而有的靠后，而人类打字的时候一般对单词的前面部分比较确定。我们对这个列表中具有相同分数的子列表进行二次排序，使得错拼位置靠后的单词排序靠前，这样会比较符合实际情况。

```
# sort by score
result_list = sorted(result_dict.items(), key=lambda x: x[1], reverse=True)
result_list = result_list[:50]

# Sort again
optimal = []
len_opt = 0
for i in range(len(result_list)):
    if result_list[i][1] != result_list[0][1]:
        break
    optimal.append([result_list[i][0], 0])
    len_opt += 1

# sort the optimal by likelihood
for i in range(len(optimal)):
    optimal[i][1] = self.cmp(word, optimal[i][0])
optimal.sort(key=lambda x: x[1], reverse=True)

# the final list
final = [x[0] for x in optimal]
resdue = [x[0] for x in result_list[len_opt:]]
final.extend(resdue)
```

- 部分词典比对。
  词典中的单词数量一般非常多，如果直接将每一个单词都拿来与错拼单词进行最长公共子串的求解，则非常耗时，而且也没有必要，因为我们在打字的过程中很难将一个单词错打为一个比它短得多的序列，或者是比它长得多的单词，因此这个技巧是有一定道理的。在这里我们使用了一个非常简单的加速技巧，就是只将长度和错拼单词相差为 N 的单词作匹配，在代码的具体实现中，我们取 $N = 2$。

```
nw = NWMatcher()
result_dict = {}

for item in self.dict:
    if not m - 2 < len(item) < m + 2:
        continue
    result_dict[item] = nw.match(word, item)
```

6 代码运行简述

- 使用概述：
  我们的自动纠错系统 BLDSC 使用 python3 代码写成，当安装完成相应的依赖包后就可以进行测试。代码运行时，获取指定的输入文件，该输入文件应该是包含单词拼写错误的英文文本，而且字符编码必须是 *UTF-8* 格式。首先我们会将文本中的所有单词进行检测并保存到一一个列表中，通过检错和纠错算法循环地对每一个单词进行处理，完成所有单词的处理之后，将每个错误单词对应的建议单词写入到原文本中相应错误单词的后面（用括号分隔表示，同时在错误单词前面用 # 号标记）。由于 python 是解释性语言，整个过程对于较大的文本可能会有一点儿延时。

- 环境要求：
  python3 编程环境，包含 **optparser** 和 **numpy** 包。

- 运行方法：
  进入主目录，运行
  python main.py
  仅仅通过上述方法调用将默认纠错建议词数为 1，同时将使用主目录下的 *test.txt* 文件作为输入，而纠错的结果将自动输出到 *result.txt* 文件中。为了可以自由地定义输入和输出文件，并指定纠错建议的个数，可以通过除加命令行参数的方法进行调用，参数介绍为：
  python main.py -help

缩写	全写	默认	简介
-i	–input	test.txt	指定输入文件
-o	–output	result.txt	指定输出文件
-n	–num_advice	1	建议词的数量

  完全的调用方式为：
  python main.py -i in_file -o out_file -n 2

7 系统评测

GNU Aspell[11] 是一个免费开源的拼写检查器,它的设计初衷是为了取代已经过时的 Ispell[10], 和我们所设计的简单的 Bspell 系统一样, 它的功能也是为英文文本提供拼写检查和可能的正确单词建议。GNU Aspell 的设计者为这个系统设计了一个非常合理的性能检测方法，并与之前所出现的一些主要的拼写检查器进行了比较，在此，我们将沿用这个方法，并使用同样的测试数据，以便对我们所完成的系统进行真实合理的评测，并在和主流系统的对比中了解我们设计的框架的合理性。

7.1 测试数据

Aspell 所用的测试数据包含 546 个错误拼写的单词以及它们相应的正确形式，每个错拼单词和正确单词在文件中占一行，因此文件一共包括 546 行，文件所在的网址为`http://aspell.net/test/cur/batch0.tab`。在测试的过程中，为了排除字典文件带来的性能误差，和 Aspell 一样，我们只对其正确形式在词典中可以找到的单词进行统计，这样所涉及的误拼单词的规模对 Bspell 来说是 478，而对 Aspell 来说是 493, 我们假设这个很小的不同对评测结果没有本质上的影响。

7.2 评测标准

Aspell 的评测项共有 8 个，分别是：

- Total (T)。Total 是指测试集中误拼单词的正确形式在字典中能够查到的误拼单词集合或其大小，因为只有正确形式出现在字典中的单词，我们才有可能进行正确的预测；
- Rank1 (R1)。第一个建议预测正确的个数；
- Rank5 (R5)。前 5 个建议中预测正确的个数，以下 Rank10,Rank25,Rank50 类似；
- Total Found (TF)。Total 中的误拼单词被正确预测的个数（Rank1-Rank50 可以找到）；
- Total Not Found (TNF)。没有预测正确的单词个数；

- Score。定义为 $\frac{TotalFound}{Total} \times 100\%$。
- Time。评测过程进行的时间。

7.3 评测结果

我们按照上述写了一个简单的评测代码，对 BSpell 系统进行了评测，代码命名为 *test.py*，包含在项目目录中。评测得到的结果为：

	Score	TNF	TF	R1	R5	R10	R25	R50	Time (min)
Aspell .31 / Normal	93.1	34	459	59.6	85.6	90.1	92.9	93.1	3.58
Aspell .33 / Normal	91.7	41	452	57.0	80.9	88.0	91.1	91.5	29.46
Aspell 0.50 / Normal	91.9	40	453	57.2	81.1	88.4	91.5	91.7	24.02
Aspell 0.60 / Normal	89.0	54	439	53.1	79.7	86.2	88.4	89.0	0.69
Aspell 0.60 / Slow	93.1	34	459	53.3	80.7	89.0	92.5	93.1	4.82
Aspell 0.60.6 / Normal	89.7	51	442	53.3	80.1	86.8	89.0	89.7	1.18
Aspell 0.60.6 / Slow	92.9	35	458	53.3	80.5	88.8	92.3	92.9	5.26
Hunspell 1.1.12	81.7	90	403	55.2	78.1	80.7	81.7	81.7	24.39
Hunspell w/ Phonetic Lookup	85.6	71	422	56.0	81.9	85.2	85.6	85.6	42.16
Ispell 3.1.20 w/ -S option	54.8	223	270	38.3	51.9	54.0	54.8	54.8	0.15
Word 97	72.2	137	356	59.0	70.8	72.2	72.2	72.2	
Bspell	81.4	89	389	50.0	66.1	73.0	79.3	81.4	30.0

根据这个评测结果，我们可以看到 Bspell 系统大概和 Hunspell 1.1.12 的性能处于相同级别，相对于最新版的 Aspell 拼写检查器来说还远远落于下风。当然，Aspell 是一个已经非常成熟的拼写检查器，在众多的编辑器如 emacs 和 vim 中都可以看到它的身影，因此这个结果也是在意料之中的。

8 组员分工

8.1 包利强

参考文献

[1] S. Deorowicz and M. G. Ciura, "Correcting spelling errors by modelling their causes," *International journal of applied mathematics and computer science*, vol. 15, no. 2, p. 275, 2005.

[2] F. J. Damerau and E. Mays, "An examination of undetected typing errors," *Information Processing & Management*, vol. 25, no. 6, pp. 659–664, 1989.

[3] F. J. Damerau, "Evaluating computer-generated domain-oriented vocabularies," *Information processing & management*, vol. 26, no. 6, pp. 791–801, 1990.

[4] J. L. Peterson, "A note on undetected typing errors," *Communications of the ACM*, vol. 29, no. 7, pp. 633–637, 1986.

[5] D. E. Knuth, "The art of computer programming, vol. 3: Sorting and searching," 1973," *This book describes the binary search on*, pp. 409–426.

[6] Z. J. Czech, G. Havas, and B. S. Majewski, "Perfect hashing," *Theoretical Computer Science*, vol. 182, no. 1-2, pp. 1–143, 1997.

[7] K. Maly, "Compressed tries," *Communications of the ACM*, vol. 19, no. 7, pp. 409–415, 1976.

[8] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[9] J. J. Darragh, J. G. Cleary, and I. H. Witten, "Bonsai: a compact representation of trees," *Software: Practice and Experience*, vol. 23, no. 3, pp. 277–291, 1993.

[10] R. Gorin, P. Willisson, W. Buehring, and G. Kuenning, "2003. international ispell," 1971.

[11] K. Atkinson, "Gnu aspell. 2003," *URL http://aspell. net*, 2011.

[12] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[13] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[14] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[15] R. Baeza-Yates and G. Navarro, "Fast approximate string matching in a dictionary," in *String Processing and Information Retrieval: A South American Symposium, 1998. Proceedings*, pp. 14–22, IEEE, 1998.

[16] M. Odell and R. Russell, "The soundex coding system," *US Patents*, vol. 1261167, 1918.

[17] J. J. Pollock and A. Zamora, "System design for detection and correction of spelling errors in scientific and scholarly text," *Journal of the American Society for Information Science (pre-1986)*, vol. 35, no. 2, p. 104, 1984.

[18] E. J. Yannakoudakis and D. Fawthrop, “An intelligent spelling error corrector,” *Information Processing & Management*, vol. 19, no. 2, pp. 101–108, 1983.

[19] E. J. Yannakoudakis and D. Fawthrop, “The rules of spelling errors,” *Information Processing & Management*, vol. 19, no. 2, pp. 87–99, 1983.

[20] K. W. Church and W. A. Gale, “Probability scoring for spelling correction,” *Statistics and Computing*, vol. 1, no. 2, pp. 93–103, 1991.

[21] T. Gadd, “Phonix: The algorithm,” *Program*, vol. 24, no. 4, pp. 363–366, 1990.

[22] L. Philips, “Hanging on the metaphone,” *Computer Language*, vol. 7, no. 12 (December), 1990.

[23] L. Philips, “The double metaphone search algorithm,” *C/C++ users journal*, vol. 18, no. 6, pp. 38–43, 2000.

[24] I. Holyer, “The np-completeness of edge-coloring,” *SIAM Journal on computing*, vol. 10, no. 4, pp. 718–720, 1981.