Shanghai Jiao Tong University

CS124

Final Report (Project B)

Chinese Word Segmentation

Team Name: Smart Boys
Team Leader: Zhuyang Wang
Team Member: Pihe Hu, Chaoran Xu
January 8, 2016

Contents

1 Prototype System Introduction **2**
- 1.1 Functions . . . 2
- 1.2 Running Environment . . . 2
- 1.3 Developing Environment . . . 2
- 1.4 Install and Run . . . 2

2 Task Allocation **3**

3 Core Algorithm **3**
- 3.1 Overview . . . 3
- 3.2 First Layer . . . 3
  - 3.2.1 Intro . . . 3
  - 3.2.2 Matching Algorithm(Complex Matching) . . . 4
  - 3.2.3 Disambiguation Algorithm . . . 6
  - 3.2.4 Conclusion . . . 14
- 3.3 Second Layer . . . 14
  - 3.3.1 Hidden Markov Model . . . 14
  - 3.3.2 Viterbi Algorithm . . . 15
- 3.4 Lexicon . . . 16

4 User Interface Component **19**
- 4.1 Procedure . . . 19
- 4.2 Programs . . . 20
- 4.3 Screenshots . . . 21

5 Testing Result **22**
- 5.1 Screenshots . . . 22

6 Conclusion **22**

7 Reference **23**

1 Prototype System Introduction

1.1 Functions

Smart Segmentation can divide the given material into sentence, and then implement word segmentation. It can recognize unlisted word to some extent. User can also modify the lexicon, add new words and delete old words. All these operation can be accomplished in GUI.

1.2 Running Environment

- Windows 7
- Windows 8.1
- Arch Linux

1.3 Developing Environment

- PyScripter
- Python IDLE
- Vim
- Qt Designer

1.4 Install and Run

GUI Dependence: PyQt4.

GUI: Under the directory, run `main.py`

CLI: `import segment`, then use the function `main(string)`

2 Task Allocation

Chaoran Xu: User Interface Component

Pihe Hu: First layer of the segmentation, using MMSEG

Zhuyang Wang: Second layer of the segmentation, using HMM; Lexicon based on Trie

3 Core Algorithm

3.1 Overview

The whole system is composed of two layers. First layer use MMSEG and a lexicon to do the initial segmentation. As for the unlisted word, the second layer is to deal with it, of which the algorithm is based on Hidden Markov Model.

3.2 First Layer

3.2.1 Intro

The first layer of the Chinese segmentation is MMSEG, which is a word identification system for mandarin Chinese text. The system consists of one lexicon, one matching algorithms and one disambiguation algorithm which is formed by four ambiguity resolution rules.

As we all known, the general used Chinese segmentation algorithm based on lexicon matching is by scanning the text from one direction pick out the longest word that it's in the given lexicon, only with a accuracy rate of 70-80% with too much ambiguities that are not acceptable for a further use. However, these conclusions can be yielded from a variety of matching tests:

1. The accuracy rate of scanning from left to right is higher than that from the negative direction;
2. The maximum multiple-character chunk is likely to be the right separation.

Based on such facts, we first find out the three-character word from the chunk, if it has. And then use ambiguity resolution rules to disambiguate.

3.2.2 Matching Algorithm(Complex Matching)

Different from the general simple matching algorithm, we use complex maximum match to find out all three-character chunks. First search the lexicon to see if the first character is a one-character word, then search this word and one that is adjacent to it and see if it is a two-character word, and so on, until all combinations are founded.

Furthermore, perform the same process to the left text until find all three-character word. Take these word, then continue this process until the last word of the string is identified. Then apply ambiguity resolution rules to find the the most possible segmentation form.

Algorithm Implementation:

The function `complex_match(chunk,result)` in mmseghmm.py is a function based on recursion algorithm that can find out all three-character word from right to left in a Chinese text, and store the partition result in a list temp which contends all possible partitions. If there is no three-character word in chunk, just pick out the possible partition.

Source Code:

```
1  def complex_match(chunk,result):
2      if len(result)==3:
3          return True
4      if len(chunk)==0:
5          return True
6      for i in range(len(chunk)):
7          if trie_search(chunk[len(chunk)-i-1:])!=-1:
8              if complex_match(chunk[:len(chunk)-i-1],
9                               [chunk[len(chunk)-i-1:]]+result)==True:
10                 temp.append([chunk[len(chunk)-i-1:]]+result)
```

Instances:

Text1:
积极开拓发达国家市场
Result:
家 市 场
国家 市 场
国 家 市场
达 国家 市场
发达国家 市 场
发达 国家 市场
拓 发达国家 市场
开拓 发达国家 市场

Text2:
开启中国推进国际产能合作大门
Result:
作 大 门
合作 大 门
合 作大 门
合 作 大门
能 合作 大门
产能 合作 大门

Text3:
帮助他们加速工业化
Result:
工 业 化
速 工业 化
加速 工业 化
加 速 工业化
们 加速 工业化
他们 加速 工业化

3.2.3 Disambiguation Algorithm

Though the results yielded from the complex matching all seem to be a possible partition according to the lexicon. There can also be lots of ambiguities due to the flexible morphemic patterns in Chinese. It's imperative to use several ambiguity resolution rules to elimilate ambiguity.

We use four ambiguity resolution rules that can almost solve about 95% of ambiguity by selecting the most possible partition forms. These four resolution rules are shown as below.

Rule One: Maximum Matching

Due to the diverse linguistic phenomena, especially almost every single Chinese character can be used as a word, we select the three-word chunk (if it cannot be separated into three parts, just take the its form, too. That is, it's not a real) that has the maximum length as the right form of partition. It is because that Chinese text is likely to use long words that can exactly express what the writer want to say.

Algorithm Implementation:

Complex maximum matching: Pick the first word from the chunk with maximum length. The related functions are `len_list(list)` in mmseg+hmm.py, which is used to calculate the length of a three-word chunk; and `largest_len(list)` in mmseg+hmm.py to pick out the chunk that has the maximum length.

Source Code:

```
1  def len_list(list):
2      string=''
3      for i in list:
4          string+=i
5      return len(string)
6
7  def largest_len(list):
8      max=list[0]
9      result=[]
10     for i in list:
11         if len_list(i)>len_list(max):
```

```
12              max=i
13              result=[]
14          elif len_list(i)==len_list(max) and i!=max:
15              result.append(i)
16      result.append(max)
17          return result
```

Instance:

Apply the rule to the result of all three-character chunk partition, we can find the most possible partition form:

Text1:
积极开拓发达国家市场
Result:
家 市 场
国家 市 场
国 家 市场
达 国家 市场
发达国家 市 场
发达 国家 市场
拓 发达国家 市场
开拓 发达国家 市场 ------ Right separation with maximum word length 8

Text2:
开启中国推进国际产能合作大门
Result:
作 大 门
合作 大 门
合 作大 门
合 作 大门
能 合作 大门
产能 合作 大门 ------ Right separation with maximum word length 6

Text3:
帮助他们加速工业化
Result:
工 业 化

速 工业 化
加速 工业 化
加 速 工业化
们 加速 工业化
他们 加速 工业化 ------ Right separation with maximum word length 7

Rule Two: Largest Average Word Length

Though rule one can really disambiguate about 80% of ambiguity successfully, some ambiguity still cannot be eliminate, such chunks can be ['城镇', '化', '进程'], ['城镇化', '进程'], ['城镇化', '进', '程'] that have the same word length and it's exactly the maximum length. So the rule two is necessary to improve the accuracy rate of a word identification system.

Rule two achieve this goal by picking the first word from the chunk with largest average word length. In the above example, it picks ['城镇化', '进程'] from the chunk. The assumption of this rule is that it is more likely to encounter multi-character words than one-character words.

Algorithm Implementation:

Use the function `largest_average_length(list)` in mmseg+hmm.py to pick out the three-character chunk that has the largest average length in chunks with the same total length.

Source Code:

```
1  def largest_average_length(list):
2      max=list[0]
3      result=[]
4      for i in list:
5          if average_word_length(i)>average_word_length(max):
6              max=i
7              result=[]
8          elif average_word_length(i)==average_word_length(max) and i!=max:
9              result.append(i)
10     result.append(max)
11     return result
```

Instance:

Text 1:
国际产能
All three-character chunk:
[['际', '产', '能'], ['国际', '产', '能'], ['国', '际', '产能'], ['国际', '产能']]
Maximum matching:
[['国', '际', '产能'], ['国际', '产能'], ['国际', '产', '能']]
Maximum matching with the largest average word length:
[['国际', '产能']]

Text 2:
实现双赢
All three-character chunk:
[['现', '双', '赢'], ['实现', '双', '赢'], ['实', '现', '双赢'], ['实现', '双赢']]
Maximum matching:
[['实', '现', '双赢'], ['实现', '双赢'], ['实现', '双', '赢']]
Maximum matching with the largest average word length:
[['实现', '双赢']]

Text 3:
塔吉克斯坦提出
All three-character chunk:
[['坦', '提', '出'], ['斯坦', '提', '出'], ['塔吉克斯坦', '提', '出'], ['斯', '坦', '提出'], ['克斯', '坦', '提出'], ['吉克斯', '坦', '提出'], ['克', '斯坦', '提出'], ['塔吉克', '斯坦', '提出'], ['塔吉克斯坦', '提出']]
Maximum matching:
[['塔吉克', '斯坦', '提出'], ['塔吉克斯坦', '提出'], ['塔吉克斯坦', '提', '出']]
Maximum matching with the largest average word length:
[['塔吉克斯坦', '提出']]

Rule Three: Smallest Variance of Word Lengths

As we can seen, that the rule two is useful only for condition in which one

or more word position in the chunks are empty. When the chunks are real three-word chunks, this rule is not useful. Because three-word chunks with the same total length will certainly have the same average length. Therefore we need another solution. There are quite a few ambiguous conditions in which the Rule 1 and Rule 2 cannot resolve.

For example, these two chunks have the same length: `[['深圳', '市委'], ['深圳市', '委']]`.And there is a rule can distinguish these two ambiguous conditions. That is, the most possible form is likely to have smaller variance of word lengths. So, rule three picks the first of the chunk with smallest variance of word lengths. In the above example, it picks `['深圳', '市委']` from the first chunk. The assumption of this rule is that word lengths are usually evenly distributed.

Algorithm Implementation:

Use the function `variance_of_word_lengths(list)` in mmseg+hmm.py to calculate the variance of word lengths of a chunk and function `largest_average_length(list)` in mmseg+hmm.pyt o pick out the three-character chunk that has the largest average length in chunks with the same total length.

Source Code:

```
1  def variance_of_word_lengths(list):
2      average_length=average_word_length(list)
3      variance=0
4      for i in list:
5          variance+=(len(i)-average_length)**2/3
6      return variance
7  def smallest_variance(list):
8      smallest=list[0]
9      result=[]
10     for i in list:
11         if variance_of_word_lengths(i)<variance_of_word_lengths(smallest):
12             smallest=i
13             result=[]
14         elif (variance_of_word_lengths(i)==variance_of_word_lengths(smallest)
15               and i!=smallest):
16             result.append(i)
17     result.append(smallest)
18     return result
```

Instances:

```
Text 1:
广东深圳市光明新区凤凰社区
Maximum matching with the largest average word length:
['新区', '凤凰社', '区'] ------ variance 0.6666666666666666
['新区', '凤凰', '社区'] ------ variance 0.0000000000000000
Maximum matching with the largest average word length and
smallest variance of word lengths:
['新区', '凤凰', '社区'] ------ variance 0.0000000000000000

Text 2:
住房城乡建设部等
Maximum matching with the largest average word length:
['城乡建设', '部', '等'] ------ variance 1.9999999999999998
['城乡', '建设部', '等'] ------ variance 0.6666666666666666
Maximum matching with the largest average word length and
smallest variance of word lengths:
[['城乡', '建设部', '等']] ------ variance 0.6666666666666666

Text 3:
深圳市委
Maximum matching with the largest average word length:
['深圳', '市委'] ------ variance 0.0000000000000000
['深圳市', '委'] ------ variance 0.6666666666666666
Maximum matching with the largest average word length and
smallest variance of word lengths:
['深圳', '市委'] ------ variance 0.6666666666666666
```

Rule Four: Largest Sum of Degree of Morphemic Freedom of Words

Cases are rare that there are more than one chunks with the same value of smallest variance of word lengths. However, it exists exactly, so apply this next rule. Such case is ['一', '定要', '认真'], and ['一定', '要', '认真'], which both have the smallest variance of word lengths.

Here we will focus on frequency of words. Chinese characters differ in their

degree of morphemic freedom. Some characters are rarely used as free morphemes, but others have larger degree of freedom. The frequency of occurrence of a character can serve as an index of its degree of morphemic freedom. A high frequency word is more likely to be used, and vice versa. Rule 4 picks the word of the chunk with largest sum of log(frequency) as the correct partition.

Algorithm Implementation:

The formula used to calculate the sum of degree of morphemic freedom is to sum log(frequency) of all word(s) in a chunk. The related function is `count_word(dic)` in mmseg+hmm.py to count the sum of total occurrence of all words in the lexicon by recursion, `sum_word_frequency(list)` in mmseg+hmm.py to calculate the sum of word frequency and function `largest_frequency(list)` to pick out the chunk with largest sum of degree of morphemic freedom of words as the correct partition.

Source Code:

```
1  def count_word(dic):
2      count=0
3      for i in dic.keys():
4          if i==0:
5              count+=dic[0]
6          else:
7              count+=count_word(dic[i])
8      return count
9
10 items=count_word(TRIE)
11
12 def sum__word_frequency(list):
13     result=0
14     for i in list:
15         result+=math.log10(trie_search(i)/items)
16     return result
17
18 def largest_frequency(list):
19     max=list[0]
20     result=[]
21     for i in list:
22         if sum__word_frequency(i)>sum__word_frequency(max):
```

```
23              max=i
24              result=[]
25          elif sum__word_frequency(i)==sum__word_frequency(max) and i!=max:
26              result.append(i)
27      result.append(max)
28          return result
```

Instance:

```
Text 1:
一定要认真
Maximum matching with the smallest variance of word lengths:
['一', '定要', '认真'] ------ sum of degree of morphemic freedom
                              of words: -13.824816563386193
['一定', '要', '认真'] ------ sum of degree of morphemic freedom
                              of words: -10.042316103214926
Maximum matching largest sum of degree of morphemic freedom of words:
['一定', '要', '认真'] ------ sum of degree of morphemic freedom
                              of words: -10.042316103214926

Text 2:
各族人民
Maximum matching with the smallest variance of word lengths:
['各', '族', '人民'] ------ sum of degree of morphemic freedom
                            of words: -10.303160088734723
['各', '族人', '民'] ------ sum of degree of morphemic freedom
                            of words: -12.180905145093616
Maximum matching largest sum of degree of morphemic freedom of words:
['各', '族', '人民']] ------ sum of degree of morphemic freedom
                             of words: -10.303160088734723

Text 3:
分析了当前
Maximum matching with the smallest variance of word lengths:
['分析', '了', '当前'] ------ sum of degree of morphemic freedom
                              of words: -9.935241087068093
['分析', '了当', '前'] ------ sum of degree of morphemic freedom
                              of words: -13.69168997010010
```

```
Maximum matching largest sum of degree of morphemic freedom of words:
['分析', '了', '当前'] ------ sum of degree of morphemic freedom
                              of words: -9.935241087068093
```

3.2.4 Conclusion

Since it is very unlikely that two characters will have exactly the same frequency value, there should be no ambiguity after this rule has been applied. So, by the complex maximum matching and four power ambiguity resolution rules we can almost solve most ambiguity with a ideal accuracy rate.

However, MMSEG is a is a word identification system that is good at normal Chinese word segmentation and relies on lexicon a lot. In other words, when faced with cases where some words are not in the lexicon, it may be difficult for it to make the correct partition, such words can be a name for a person or a place. So, we process these cases in second layer.

3.3 Second Layer

Second Layer is based on Hidden Markov Model and viterbi algorithm.

3.3.1 Hidden Markov Model

Consider a material which is already segmented, we use four tags(B, E, M, S) to label every single character:

`B` The first character of a word.(**B**egin)

`E` The last character of a word.(**E**nd)

`M` The character in the middle of a word.(**M**iddle)

`S` Single character word.(**S**ingle.)

For example:

我们是中国人

The corresponding label sequence is `BESBME`

After dealing with a large training material, we finally get a model consists of:

- the probability of a character being labeled as B, E, S or M. The result is stored in prob_emit.py
- the probability of a sentence starting with B or S. The result is stored in prob_start.py
- the probability of transform, that is, the probability of a label followed by the next corresponding label. For example, the probability of B followed by B, E, M, S. The result is stored in prob_trans.py

We train the model by the material from
http://sighan.cs.uchicago.edu/bakeoff2005/
Using msr_training.utf8 and pku_training.utf8

And the final size of prob_emit.py is 452.9 kB, which is 1.3 MB in jieba segmentation.

3.3.2 Viterbi Algorithm

Consider a sentence of n characters. Every character can be labeled as B, E, S or M. Thus finally there will be 4^n combinations. Then we only need to calculate the corresponding probability and pick out the maximum. However, 4^n is so large a number that it will cost so much time. Hence we use viterbi algorithm which is a certain type of dynamic programming.

The function `init_array(x, y)` is to provide an x×y list to simulate array.

```
1 def init_array(x, y):
2     array = []
3     for i in range(x):
4         array.append([])
5         for j in range(y):
6             array[i].append(0)
7     return array
```

And here is the core algorithm:

```
1 def viterbi(sentence):
2     length = len(sentence)
3     weight = init_array(length, 4)
```

```
    path = init_array(length, 4)
    for i in range(4):
        weight[0][i] = PROB_START[i] + PROB_EMIT[i].get(sentence[0], MINPROB)
    for i in range(1, length):
        for j in range(4):
            weight[i][j] = MINPROB
            path[i][j] = -1;
            p_emit = PROB_EMIT[j].get(sentence[i], MINPROB)
            for k in range(4):
                temp = weight[i-1][k]+PROB_TRANS[k][j]+p_emit
                if temp > weight[i][j]:
                    weight[i][j] = temp
                    path[i][j] = k
    if weight[length-1][1] > weight[length-1][3]:
        pos = 1
    else:
        pos = 3
    label = []
    table = ['B','E','M','S']
    for i in range(length-1, -1, -1):
        label.append(table[int(pos)])
        pos = path[i][pos]
    return ''.join(label[::-1])
```

The function `output(line)` will produce a list of cut word according to the path calculated by viterbi algorithm.

```
def output(line):
    split_line = ['']
    if line == '': return split_line
    label = viterbi(line)
    j = 0
    for i in range(len(line)):
        split_line[j] += line[i]
        if label[i] in ['E', 'S']:
            j += 1
            split_line.append('')
    return split_line
```

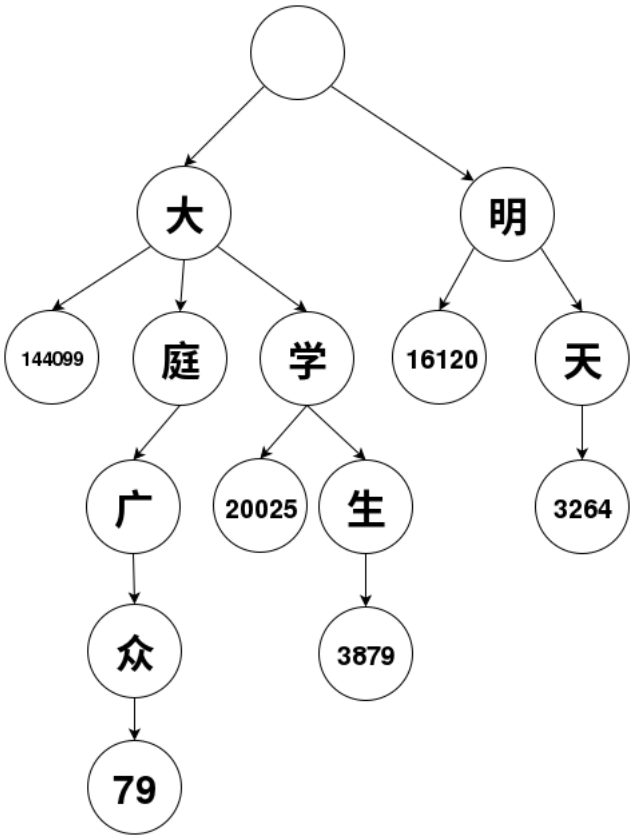
3.4 Lexicon

The lexicon is from:
http://download.csdn.net/detail/logken/3575376

and
https://github.com/fxsjy/jieba

We manually delete more than 20000 words in the lexicon.

We use trie, which is also called prefix tree, to store the lexicon. In Trie, all the descendants of a node have a common prefix associated with that node, and the root is associated with the empty string. And every node stores one Chinese character, while every leaf stores a number, which is exactly the value of its prefix. Here is an simple example below.



We first consider double array trie to achieve better performance, however, I find it's a little bit difficult because it's hard to get the exact range of Chinese character in Unicode(I only get an approximate range from `u4E00` to `u9FFF`). Hence it's hard to accurately map them.

Besides, there is no such a data structure 'array' in Python. Python's list

is more powerful than array, but just too powerful to implement the double array trie, taking much more space and had no advantage over something else. Initially I think about import the library numpy or embed C in Python, but time is limited, so I finally choose the dictionary in Python to store the trie.

Thus the corresponding dictionary of the above trie is:

```
{'大': {0: 144099, '学': {'生': {0: 3879}}, '庭': {'广':
{'众': {0: 79}}}}, '明': {0: 16120, '天': {0: 3264}}}
```

Below are three functions that can add, delete and search word in the trie. The variable `ref` is the current position in trie. `ref = ref[char]` means move from current node `ref` to its descendant `char`.

```
1  def trie_add(word, prob):
2      ref = TRIE
3      if trie_search(word) == -1:
4          for char in word:
5              if char not in ref:
6                  ref[char] = {}
7              ref = ref[char]
8          ref[0] = prob
9          return True
10     else:
11         return False
12
13
14 def trie_delete(word):
15     if trie_search(word) != -1:
16         ref = TRIE
17         for i, char in enumerate(word):
18             if len(ref[char]) == 1:
19                 del ref[char]
20                 return True
21             elif i == len(word)-1:
22                 del ref[char][0]
23                 return True
24             else:
25                 ref = ref[char]
26     else:
27         return False
28
29
30 def trie_search(word):
```

```
31      ref = TRIE
32      for char in word:
33          if char not in ref:
34              return -1
35          else:
36              ref = ref[char]
37      if 0 not in ref:
38          return -1
39      else:
40          return ref[0]
```

4 User Interface Component

4.1 Procedure

When we discussed the distribution of our work,I chose to design the user interface,then I began to make some preparation.Firstly,I read the materials which teacher Yao gave us before.I intended to use tkinter to design our user interface,but my partner told me that QT was more powerful,so I eventually chose QT.However,I had no materials about how to use QT,so I had to spent lots of time searching the Internet to find some useful information.Fortunately,I succeeded in finding some very helpful information,which could help me accomplish our project.Then I started to learn how to use QT with the assistant of my partner.

I tried to design some simple window according to the learning materials,and it turned out to be successful.Then I made out the frame of our user interface,but it had no function.To accomplish the user interface,I read some materials and added some functions step by step,but the lexicon and related functions still needed to be improved.

After several failure,I added some buttons to the lexicon to add some useful functions.Now our user interface can perform segmentation and make operations on the lexicon.For example,you can choose to separate an article into sentences or separate a sentence into words by click related buttons.What's more,you can open a file instead of input the content word by word.You can also save the input or output as a new file.

Moreover,our users are able to see our lexicon,which will be showed in a new window.There are also several useful buttons in the new window,the users can add new words to the lexicon,delete words from the lexicon,sort the words in the lexicon,undo their operations and save the lexicon.The operations are very easy because users only need to click the related buttons.

4.2 Programs

Files that are generated by QtDesigner are in the directory windows.

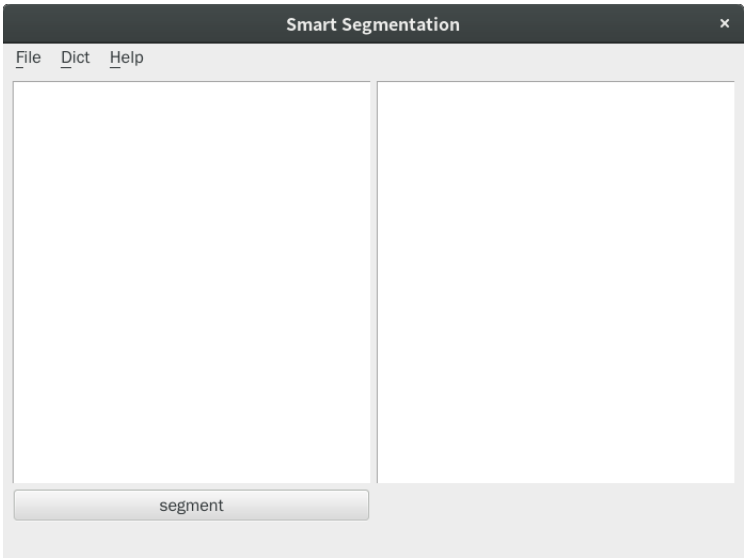
The main program is main.py.

The most complicated part is the lexicon window. When user type some character in the search line, it will send a signal `textEdited` and call the function `search`. If the current word is in the lexicon, the dict viewer will automatically jump to the corresponding word, so the user don't need to click a search button or something else, they just need to type and can see the result immediately. And when user click the word in the left lexicon viewer, the word will appear in the search line, so the user can click some button to operate.

Another useful function is undo. I use a list called `undo_list` to store the sequence of operations, and it is a stack for sure. I use a flag to represent whether the word is added or deleted. So the stack stores pairs of word and flag.

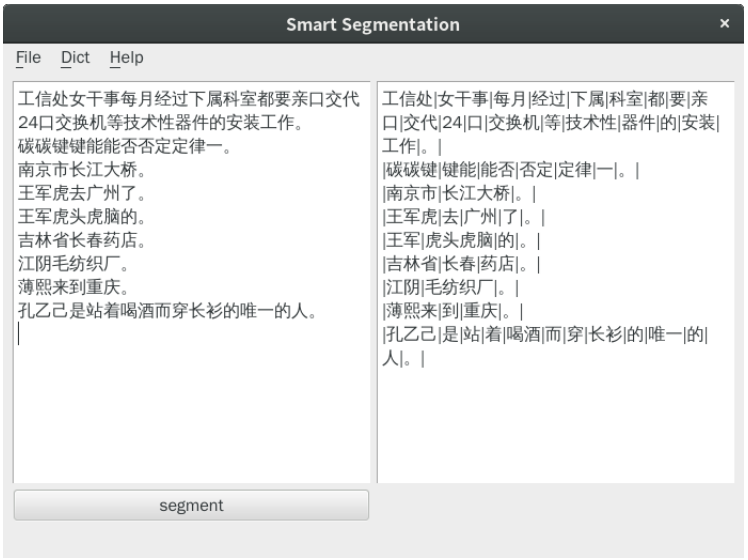
Moreover, to prevent the user's misoperation, I use `QListWidget` to display the lexicon, and all the operation can only accomplished by the related buttons. What's more, when one operation cannot be accomplished, the related button cannot be press, so the logic of operation is more clear.

4.3 Screenshots



5 Testing Result

5.1 Screenshots



6 Conclusion

Through this final project, we broaden our horizons, get to see the beauty of computer science, natural language processing and Python, and enhance our ability of programming. Below are some important lessons we have learned:

Teamwork The final project is accomplished by our three, everyone is indispensable. Only by teamwork can we finally finish this final project. And it needs communication, proper task allocation and trust in each other.

Self-learning The final project, natural language processing, is a brand new field to all of us. In the process, we look up a lot of materials, books, papers, and some open source implementation of Chinese language segmentation, as well as the tutorial of PyQt4. By the way, our ability of English reading and writing has also been improved a lot.

Modularity We realize the importance of modularity during the process of cooperation. So we decide to write simple parts connected by clean interface, considering that clarity and simplicity are more important than complexity. This is exactly the Unix philosophy: do one thing and do it well. Keep it simple and stupid. And it brings us many benifits such as the convenience to modify the program or add a new function.

Version Control We use git to control the version of our project, and you can see it on Github. I admit it is a little inconvenient to use git, however, it guarantees the project won't be lost when suddenness occurs, and we can easily return to certain stage of development when something goes wrong.

7 Reference

http://technology.chtsai.org/mmseg/
http://sighan.cs.uchicago.edu/bakeoff2005/ https://github.com/fxsjy/jieba
https://en.wikipedia.org/wiki/Trie
https://en.wikipedia.org/wiki/Hidden_Markov_model
https://en.wikipedia.org/wiki/Viterbi_algorithm
http://yanyiwu.com/work/2014/04/07/hmm-segment-xiangjie.html
http://www.52nlp.cn/
https://wiki.python.org/moin/PyQt/Tutorials
http://pyqt.sourceforge.net/Docs/PyQt4/index.html
http://stackoverflow.com/