# Practice of Part-of-Speech Tagging Based on HMM
# 基于隐马尔可夫模型的单词词性标注实践

未央-水木01　郑懿　2020012859

zhengyi20@mails.tsinghua.edu.cn

## Abstract

自然语言处理近年来发展迅速，词性标注是其中的一个重要子问题。自然语言处理中的许多问题可通过概率模型获得较好解答，马尔可夫链及其衍生模型也成为了其中一种重要工具。本文从马尔可夫链出发对单词词性标注进行建模，并通过隐马尔可夫模型（*HMM*）和*Viterbi*算法对词性标注问题进行了实践。

## 1. 模型的建立

词性（part-of-speech）是句子中各个词汇的语法分类，词性标注即为句子中的每一个单词选择其最为恰当的词性。英语中各个单词之间具有空格而中文则是连续的句子，要想对中文句子中的单词进行词性标注首先需要对句子进行切割，考虑到操作难度，笔者选取英语单词作为词性标注的对象。然而有许多单词具有多义性，以常见的单词"can"为例，它就拥有以下常见的三个词性：

- 情态动词（表“能够”），I **can** finish my homework.
- 名词（表“罐头”），I bought a fish **can**.
- 动词（表“将...装入罐头”），The vegetables are **canned** and shipped.

这正是标注的难度所在。

### 1.1. 马尔可夫链

稍加思考便不难发现，一句句子中的前后词之间是存在联系的，后一词可能的词性在很大程度上受前一词词性的影响，例如一个形容词后有很大可能接一个名词，而只有很小的可能接一个形容词。用条件概率的写法可以如下表示（数值为笔者的估计值，仅作为可能性大小的直观表达）：

$$P(n.|adj.) = 0.8, P(adj.|adj.) = 0.1$$

于是笔者尝试构建了马尔可夫链的模型，假设后一词词性与前一词词性之间存在状态转移概率的依赖条件，所有词性的集合构成了状态集合$S$，根据经验画出的状态转移概率图如下所示：

为了模型的完整性，还需给出初始概率分布，即每句句子中第一个单词是这一词性的概率，由经验给出的表格如下：

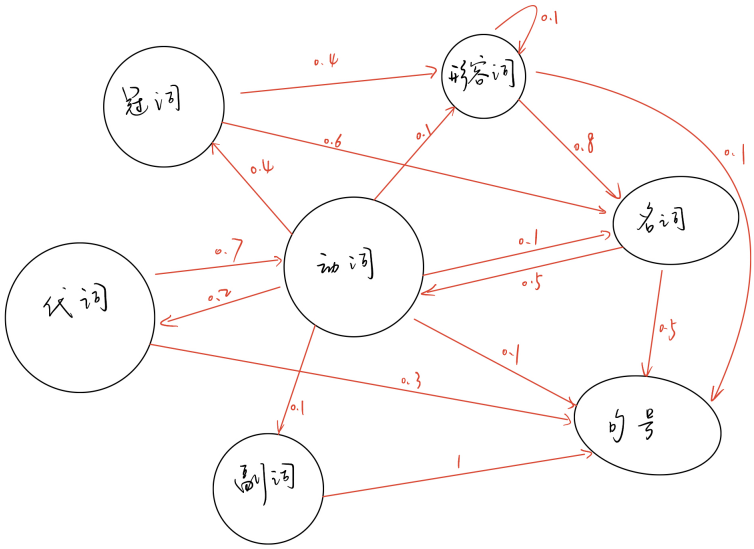


图 1. 以六种词性和句号为例的状态转移概率图

表 1. 初始概率分布表

| 冠词 | 形容词 | 代词 | 名词 | 动词 | 副词 | 句号 |
|---|---|---|---|---|---|---|
| 0.3 | 0.1 | 0.4 | 0.2 | 0 | 0 | 0 |

根据这一模型，不难给出某一英语句子词性序列为一特定词性序列的概率，例如出现“冠词-名词-动词-名词-句号”这一词性序列（符合这一词性序列的句子如“The boy drinks water.”）的概率可以如下计算：

$$\begin{aligned} & P(art.)P(n.|art.)P(v.|n.)P(n.|v.)P(.|n.) \\ =& 0.3 \times 0.6 \times 0.5 \times 0.1 \times 0.5 \\ =& 4.5 \times 10^{-3} \end{aligned}$$

用一般化的数学语言可以给出如下表述：给定状态集合

$$S = \{art., adj., pron., n., v., adv., period(\text{句号})\}$$

对于任意给定的词性序列

$$t_1 t_2 \cdots t_n, t_i \in S(i = 1, 2, \cdots, n)$$

一个英语句子的词性序列就等于该词性序列的概率为：

$$P\left(t_1 t_2 \cdots t_n\right) = P\left(t_1\right) \prod_{i=1}^{n-1} P\left(t_{i+1} \mid t_i\right)$$

其中$P(t_1)$为第一个词的词性是$t_1$的概率，$P(t_j|t_i)$为词性状态从$t_i$到$t_j$的转移概率。

需要指出的是，上述理论成立的前提是英语句子中的词性序列严格满足一阶马尔可夫性质，即后一词词性完全由前一词词性所决定，这样的假设是存在缺陷的。例如，如果形容词前为一冠词，则形容词后为名词的概率大大提高，而为句号的概率大大降低。为了改善这一情况，事实上可以采用阶数更高的马尔可夫链，即考虑相邻两

个单词的词性状态二元向量$(t_{i-1}, t_i)$，不过这样做会使得时间复杂度大大上升。考虑到篇幅的限制和模型的简洁性，这里仍然采用一阶马尔可夫链模型，在后面可以看到，一阶模型尽管存在理论上的不严谨之处，但其效果还是尚能令人满意的。

### 1.2. 隐马尔可夫模型

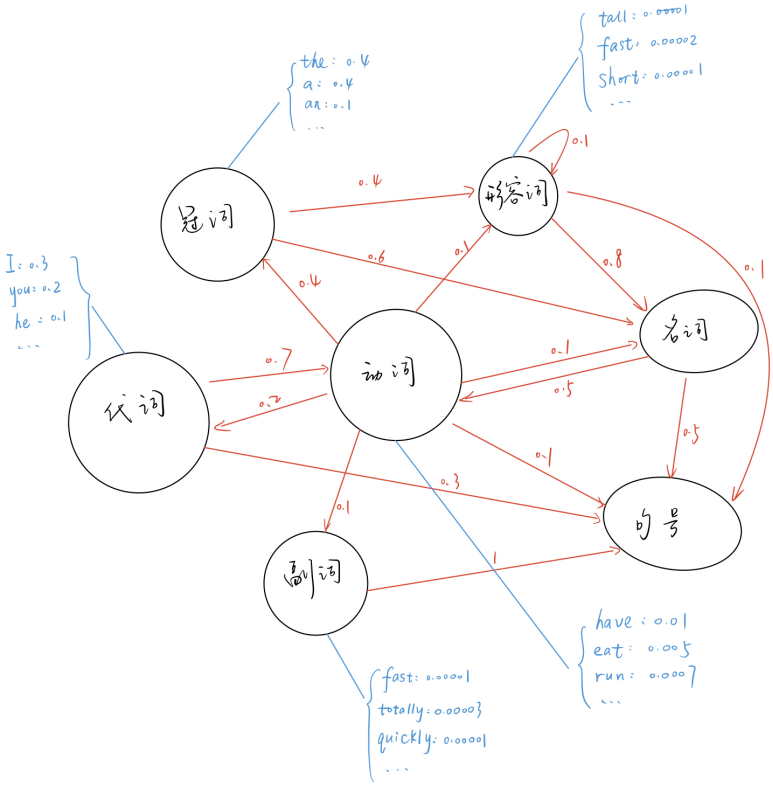


图 2. 隐马尔可夫模型下的状态转移概率图

隐马尔可夫模型（Hidden Markov Model）相比马尔可夫链，增加了**可观测符号集**（set of observable symbol）和从状态到符号的**输出概率**（emission probability）。新的状态转移概率图仍然包含原有的马尔可夫链，但现在每一个状态（词性）都可以以一定概率输出给定的单词。这里的概率是一种条件概率，是指给定词性的条件下该单词是某个特定单词的概率，即前面提到的**输出概率**；而所有单词的集合就构成了前面提到的**可观测符号集**；HMM的状态，也即词性，通常称作**隐藏状态**（hidden states）。例如，图中形容词输出“fast”的概率为0.00002，即一个词在句子中如果作为形容词，则它可能是“fast”的概率是0.00002（这里的数值也为笔者的估计值，仅供参考），可表示为

$$P(fast|adj.) = 0.00002$$

值得注意的是，**不同的状态可以输出同一个单词**，如“fast”可以作形容词也可以作副词。

建立HMM模型后，不难计算给定词性序列

$$t_1 t_2 \cdots t_n$$

后，特定句子

$$w_1 w_2 \cdots w_n$$

出现的概率，由乘法公式有：

$$\begin{aligned}&P(w_1w_2\cdots w_n|t_1t_2\cdots t_n)\\=&P(w_1|t_1)P(w_2|t_2)\cdots P(w_n|t_n)\\=&\prod_{i=1}^{n}P(w_i|t_i)\end{aligned}$$

例如，给定一词性序列为“冠词-名词-动词-冠词-名词”，句子为“The boy took the test.”的概率可以如下计算：

$$\begin{aligned}&P(The\,boy\,took\,the\,test|art.-n.-v.-art.-n.)\\=&P(The|art.)P(boy|n.)P(took|v.)P(the|art.)P(test|n.)\end{aligned}$$

同样需要指出的是，这一模型虽然在语法上是合理且完备的，但并没有考虑语义的因素，仍以词性序列“冠词-名词-动词-冠词-名词” 为例，稍加思考就容易得到这样的结论：句子为“The boy took the test.”的概率和句子为“The test took the boy.”的概率是一样的。然而，后者表达的意思是完全不合理的，即它在给定词性序列的条件下出现的概率应当远小于前者，但在这一模型中却会得到两者概率相等的结果，这亦是此模型的不足之处。但考虑到模型的简洁性，此处我们仍假设承认这一模型的合理性。

## 2. Viterbi算法的引入

我们可以这样理解词性标注问题：在给定一个单词序列$w_1w_2\cdots w_n$和一个隐马尔可夫模型的情况下，求其最可能对应的词性序列$t_1t_2\cdots t_n$，即使得$P(t_1t_2\cdots t_n|w_1w_2\cdots w_n)$最大，由Bayes公式不难得到：

$$\begin{aligned}&P(t_1t_2\cdots t_n|w_1w_2\cdots w_n)\\=&\frac{P(w_1w_2\cdots w_n|t_1t_2\cdots t_n)P(t_1t_2\cdots t_n)}{P(w_1w_2\cdots w_n)}\end{aligned}$$

此处的$P(w_1w_2\cdots w_n)$可以认为是一常数而不予考虑，因此我们只需最大化$P(w_1w_2\cdots w_n|t_1t_2\cdots t_n)P(t_1t_2\cdots t_n)$即可，而这一概率的计算公式已经给出，即：

$$\begin{aligned}&P(w_1w_2\cdots w_n|t_1t_2\cdots t_n)P(t_1t_2\cdots t_n)\\=&P\left(t_1\right)\prod_{i=1}^{n-1}P\left(t_{i+1}\mid t_i\right)\prod_{i=1}^{n}P(w_i|t_i)\\=&P\left(t_1\right)P\left(w_1\mid t_1\right)\prod_{i=2}^{n}P\left(w_i\mid t_i\right)P\left(t_i\mid t_{i-1}\right)\end{aligned}$$

如果采用遍历的方法求这一概率的最大值，那么即使是对于序列长$n=10$和总状态数$s=10$的情形，所有可能的序列数就多达$s^n=10^{10}$，这样的计算复杂度是巨大的，因此需要采用**动态规划**的做法，在这里引入Viterbi算法。

我们定义函数$v(i,t_k)$为给定$t_i$的取值为$t_k$的情况下，$P(w_1w_2\cdots w_i|t_1t_2\cdots t_i)P(t_1t_2\cdots t_i)$能够取到的最大值（此处的$t_k$是一特定值，而非词性序列中的某一元素）；再定义$bt(i,t_k)$为 $P(w_1w_2\cdots w_i|t_1t_2\cdots t_i)P(t_1t_2\cdots t_i)$取得最大值时的$t_{i-1}$的值。那么，$v(1,t_k)$按照定义应当取为$P(w_1|t_k)P(t_k)$，而除了$i=1$外的所有$i$的取值，我们都可以按照如下的递推式进行前向计算：

$$\begin{aligned}v\left(i,t_k\right)&=P\left(w_i\mid t_k\right)\cdot\max_{t_j}[v\left(i-1,t_j\right)P\left(t_k\mid t_j\right)]\\bt\left(i,t_k\right)&=\arg\max_{t_j}[v\left(i-1,t_j\right)P\left(t_k\mid t_j\right)]\end{aligned}$$

可以发现，如果计算出$v(n,t_k)$在所有$t_k$可能取值下的值，那么我们所求的概率最大值即为

$$\max_{t_k} v(n,t_k)$$

且有$t_n$的最优取值

$$t_n^* = \arg\max_{t_k} v(n,t_k)$$

而希望求得的词性序列可以通过如下反向回溯递归求得：

$$t_{n-1}^* = bt(n,t_n^*),\cdots,t_1^* = bt(2,t_2^*)$$

上述算法即为Viterbi算法，其时间复杂度为$O(ns^2)$，相比$s^n$的复杂度有了巨大的改善。

## 3. 隐马尔可夫模型的训练

笔者采用了Rice University的标注了词性的英语文本作为HMM模型的训练数据集，令 $C(t_i,w_j)$ 为训练数据中单词 $w_j$ 被标记成 $t_i$ 的次数， $C(t_{i-1},t_i)$ 为词性$t_{i-1},t_i$相继出现的次数， $C(t_i)$ 为词性$t_i$ 出现的次数， $C_1(t_i)$ 为 $t_i$ 作为句子中的第一个标记出现的次数。根据概率论中的频数作商的方法计算概率，我们有

$$P(w_i \mid t_i) = \frac{C(t_i,w_i)}{C(t_i)} \quad P(t_i) = \frac{C_1(t_i)}{\sum_i C_1(t_i)} \quad P(t_i \mid t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})}$$

这三组数据可以提供完整的HMM模型描述。

需要指出的是，无论多么庞大的训练集都难免会出现测试数据集中包含训练集中未曾出现的单词，这样的情况下模型会给出这句句子出现的概率为零的结果，从而使得Viterbi算法失效，因此，笔者在正式测试之前对测试数据集进行了预处理，找出训练集中未包含的单词，并以一个极小的概率使得每一个状态都能输出这个单词，从而使得模型可以正常运行。

## 4. 评价与思考

为了提供更加客观的评价结果，笔者又尝试了另一种朴素算法，并希望对这两种方法进行比较。正如笔者在本文开头提到的，大部分的英语单词都只有一个词性，新的方法就采用了这一特点：对于测试数据中的每一个单词$w_i$，找出它在训练时最常出现的词性$t_i^{max}$，并直接进行输出，完全不考虑前后单词间的关系。这一算法看起来远没有HMM模型可靠，但其复杂度只有$O(n)$之小。笔者测试了同一段长为974个单词的测试数据，得到的结果如下：

| | Viterbi算法 | 朴素算法 |
|---|---|---|
| 测试数据单词数 | 974 | 974 |
| 正确标记数 | 945 | 910 |
| 正确率 | 97.0226% | 93.4292% |

可以发现，HMM模型和Viterbi算法给出了较令人满意的97%的准确率，但令人惊讶的是看起来不可靠的朴素算法竟然也达到了93%的准确率，用更高的复杂度仅换来4%的准确率优势，这似乎有些不能让人接受。笔者认为，词性标注作为自然语言处理方向比较基础和简单的问题，一个比较简单的模型有较高的准确率也不是特别令人奇怪，更重要的是，当模型的准确率接近100%时，比较准确率提升的绝对百分比就显得不那么有意义了，例如99.99%和99%的准确率，尽管只相差不到1%，但它们是有天壤之别的。考虑词性标注的两个模型的误差率，前者为2.98%，后者为6.57%，前者不到后者的一半，这已经是一个不小的进步了。

此外，引用唐宏岩老师在课上提过的George Box的一句话，“All models are wrong, but some are useful.”正如笔者前面所提到的，在建立模型的过程中有许多不完善之处，例如可以将一阶马尔可夫链改为二阶甚至更高阶来提高模型精确度，但那会不可避免地带来更高的时间复杂度。在解决实际问题时，并不是越精确越好，而是“够用”就好，这也正是建立模型的意义所在。

## 5. 参考文献

[1]Wicaksono, A. F., & Purwarianti, A. (2010, August). HMM based part-of-speech tagger for Bahasa Indonesia. In Fourth International MALINDO Workshop, Jakarta.

[2]D. Jurafasky and J.H. Martin, 2009. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Second Edition. Pearson, Prentice-Hall.

[3]Lou, H-L. ”Implementing the Viterbi algorithm.” IEEE Signal processing magazine 12.5 (1995): 42-52.

# Supplementary

# 附录

## A. Viterbi算法代码实现

```cpp
#include <bits/stdc++.h>

using namespace std;

string training_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\training.txt";
string tagged_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\testdata_tagged.txt";
string untagged_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\testdata_untagged.txt";

// # of word in training data
int token_count = 0;
// # of different tags in training data
int tag_count = 0;
// maps tag_id to tag
string tags[50];
// maps tag to its id
map<string, int> id;

//initial distribution
double pi[50] = {0.01};

//transition matrix
double transition[50][50];

//emission matrix
map<string, double> emission[50];

//set of all words
set<string> dictionary;

void preprocess_training() {
    //add all words to dictionary
    //label the tags
    //count tags
    fstream training(training_path);
    string word, tag;
    while (training >> word) {
        dictionary.insert(word);
        training >> tag;
```

```
        training >> tag;
        if (id.find(tag) == id.end()) {
            tags[tag_count] = tag;
            id[tag] = tag_count;
            tag_count++;
        }
        token_count++;
    }
    training.close();
}

void process_training() {
    fstream training(training_path);
    bool first = true;
    string word, last_tag, tag;
    while (training >> word) {
        training >> tag;
        training >> tag;
        //count emission
        emission[id[tag]][word]++;
        if (first) {
            //count initial distribution
            pi[id[tag]]++;
            first = false;
        } else {
            //count transition
            transition[id[last_tag]][id[tag]]++;
        }
        last_tag = tag;

        if (word == ".")
            first = true;
    }
    training.close();
}


void normalize_model() {
    //normalize pi
    double denom = 0.0;
    for (int i = 0; i < tag_count; ++i) {
        denom += pi[i];
    }
    for (int i = 0; i < tag_count; ++i) {
        pi[i] /= denom;
```

```
84      }
85      //normalize transition
86      for (int i = 0; i < tag_count; ++i) {
87          denom = 0.0;
88          for (int j = 0; j < tag_count; ++j) {
89              denom += transition[i][j];
90          }
91          for (int j = 0; j < tag_count; ++j) {
92              transition[i][j] /= denom;
93          }
94      }
95      //normalize emission
96      for (int i = 0; i < tag_count; ++i) {
97          denom = 0.0;
98          for (auto it = emission[i].begin(); it != emission[i].end(); it++) {
99              denom += it->second;
100         }
101         for (auto it = emission[i].begin(); it != emission[i].end(); it++) {
102             it->second /= denom;
103         }
104     }
105 }
106
107 void preprocess_test() {
108     fstream test(untagged_path);
109     string word;
110     vector<string> new_word;
111     while (test >> word) {
112         if (dictionary.find(word) == dictionary.end()) {
113             //unknown word found
114             new_word.push_back(word);
115             dictionary.insert(word);
116         }
117     }
118     for (int i = 0; i < tag_count; ++i) {
119         for (string str : new_word) {
120             emission[i][str] = 0.0;
121         }
122         for (auto it = emission[i].begin(); it != emission[i].end(); it++) {
123             it->second += 0.00001;
124         }
125     }
126     //normalize emission
127     for (int i = 0; i < tag_count; ++i) {
128         double sum = 0.0;
```

```
        for (auto it = emission[i].begin(); it != emission[i].end(); it++) {
            sum += it->second;
        }
        for (auto it = emission[i].begin(); it != emission[i].end(); it++) {
            it->second /= sum;
        }
    }
    test.close();
}

int viterbi(vector<string> &word, vector<string> &tag) {
    int K = tag_count, L = word.size();
    double v[L][K];
    int bt[L][K];
    for (int i = 0; i < K; ++i) {
        v[0][i] = log(pi[i]) + log(emission[i][word[0]]);
    }
    for (int i = 1; i < L; ++i) {
        for (int j = 0; j < K; ++j) {
            v[i][j] = log(emission[j][word[i]]) + v[i - 1][0] + log(transition[0][j]);
            bt[i][j] = 0;
            for (int k = 0; k < K; ++k) {
                if (v[i][j] < log(emission[j][word[i]]) + v[i - 1][k] + log(transition[k][j
                    ])) {
                    v[i][j] = log(emission[j][word[i]]) + v[i - 1][k] + log(transition[k][j
                        ]);
                    bt[i][j] = k;
                }
            }
        }
    }
    vector<int> z(L);
    z[L - 1] = 0;
    for (int i = 0; i < K; ++i) {
        if (v[L - 1][i] > v[L - 1][z[L - 1]]) z[L - 1] = i;
    }
    for (int i = L - 2; i >= 0; --i) {
        z[i] = bt[i + 1][z[i + 1]];
    }
    int correct = 0;
    for (int i = 0; i < L; ++i) {
        if (tag[i] == tags[z[i]]) correct++;
    }
    return correct;
}
```

```
172
173 void running_test() {
174     fstream test(tagged_path);
175     string word, tag;
176     vector<string> sword, stag;
177     int total=0, correct=0;
178     while (test >> word) {
179         test >> tag;
180         test >> tag;
181         //compute emission
182         sword.push_back(word);
183         stag.push_back(tag);
184         if (word == ".") {
185             total += sword.size();
186             correct += viterbi(sword, stag);
187             sword.clear();
188             stag.clear();
189         }
190     }
191     test.close();
192     cout << "total: " << total << endl;
193     cout << "correct: " << correct << endl;
194     cout << "percent: " << 1.0 * correct / total << endl;
195 }
196
197 int main() {
198     preprocess_training();
199     cout << "training preprocessed" << endl;
200     process_training();
201     cout << "training completed" << endl;
202     normalize_model();
203     cout << "model normalized" << endl;
204     preprocess_test();
205     cout << "test preprocessed" << endl;
206     running_test();
207
208     return 0;
209 }
```

## B. 朴素算法实现

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
```

```
4
5  string training_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\training.txt";
6  string tagged_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\testdata_tagged.txt";
7  string untagged_path = "D:\\Codefield\\CODE_Cpp\\postagging-main\\testdata_untagged.txt";
8
9  // # of word in training data
10 int token_count = 0;
11 // # of different tags in training data
12 int tag_count = 0;
13 // maps tag_id to tag
14 string tags[50];
15 // maps tag to its id
16 map<string, int> id;
17
18 //frequency of each tag
19 int fr[50] = {0};
20
21 //the most frequent tag
22 int best_tag = 0;
23
24 //w_i->t_j count
25 map<string, vector<int>> emission;
26 //most frequent tag for each word
27 map<string, int> best_emission;
28
29 void preprocess_training() {
30     fstream training(training_path);
31     string word, tag;
32     while (training >> word) {
33         if (emission.find(word) == emission.end()) {
34             emission[word] = vector<int>(50, 0);
35         }
36         training >> tag;
37         training >> tag;
38         if (id.find(tag) == id.end()) {
39             tags[tag_count] = tag;
40             id[tag] = tag_count;
41             tag_count++;
42         }
43         fr[id[tag]]++;
44         token_count++;
45     }
46     for (int i = 0; i < tag_count; ++i) {
47         if (fr[best_tag] < fr[i]) best_tag = i;
48     }
```

```
49      training.close();
50  }
51
52  void process_training() {
53      fstream training(training_path);
54      string word, tag;
55      while (training >> word) {
56          training >> tag;
57          training >> tag;
58          //count emission
59          emission[word][id[tag]]++;
60      }
61      for (auto it = emission.begin(); it != emission.end(); it++) {
62          int best = 0;
63          for (int i = 0; i < it->second.size(); ++i) {
64              if (it->second[best] < it->second[i]) best = i;
65          }
66          best_emission[it->first] = best;
67      }
68      training.close();
69  }
70
71  void running_test() {
72      fstream test(tagged_path);
73      string word, tag;
74      int total = 0, correct = 0;
75      while (test >> word) {
76          test >> tag;
77          test >> tag;
78          if (best_emission.find(word) != best_emission.end()) {
79              //word seen before
80              correct += (best_emission[word] == id[tag] ? 1 : 0);
81          } else {
82              //new word
83              correct += (best_tag == id[tag] ? 1 : 0);
84          }
85          total++;
86      }
87      test.close();
88      cout << "total:␣" << total << endl;
89      cout << "correct:␣" << correct << endl;
90      cout << "percent:␣" << 1.0 * correct / total << endl;
91  }
92
93  int main() {
```

```
    preprocess_training();
    cout << "training preprocessed" << endl;
    process_training();
    cout << "training completed" << endl;
    running_test();

    return 0;
}
```

## C. 数据集（部分）

```
//training set
The / DT
final / JJ
major / JJ
items / NNS
of / IN
New / NNP
Deal / NNP
legislation / NN
were / VBD
the / DT
creation / NN
of / IN
the / DT
United / NNP
States / NNPS
Housing / NNP
Authority / NNP
and / CC
Farm / NNP
Security / NNP
Administration / NNP
, / ,
both / DT
in / IN
1937 / CD
, / ,
and / CC
the / DT
Fair / NNP
Labor / NNP
Standards / NNP
Act / NNP
```

```
of / IN
1938 / CD
, / ,
which / WDT
set / VBP
maximum / NN
hours / NNS
and / CC
minimum / NN
wages / NNS
for / IN
most / JJS
categories / NNS
of / IN
workers / NNS
. / .
```

```
// test set
The / DT
New / NNP
Deal / NNP
was / VBD
a / DT
series / NN
of / IN
domestic / JJ
programs / NNS
enacted / VBN
in / IN
the / DT
United / NNP
States / NNPS
between / IN
1933 / CD
and / CC
1936 / CD
, / ,
and / CC
a / DT
few / JJ
that / WDT
came / VBD
later / RB
. / .
```