

成绩	
----	--



**北京理工大学**  
BEIJING INSTITUTE OF TECHNOLOGY

# 自然语言理解初步

## 大作业一报告

题    目： 分词与词性标注

学    院： 计算机学院

专业名称： 计算机科学与技术

学    号： 1120191600

姓    名： 张弛

任课教师： 汤世平

评  阅  人：

# 课程大作业一：分词与词性标注

## 一、要求

- 实现基于词典的分词方法和统计分词方法：两类方法中实现一种即可；
- 对分词结果进行词性标注，也可以在分词的同时进行词性标注；
- 对分词及词性标注结果进行评价，包括 4 个指标：正确率、召回率、F1 值和效率。

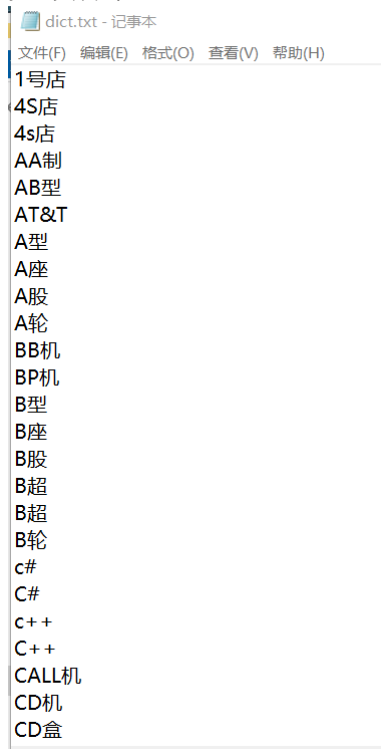
## 二、选做

命名实体识别：对句子中出现的人名、地名、机构名进行识别。

## 三、实现过程描述：

### 1. 语料库的描述与处理：

分词的词典采用网络上下载的中文分词词库汇总的词典，词典为 `data/dict.txt`，如下所示：



dict.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

1号店  
4S店  
4s店  
AA制  
AB型  
AT&T  
A型  
A座  
A股  
A轮  
BB机  
BP机  
B型  
B座  
B股  
B超  
B超  
B轮  
c#  
C#  
c++  
C++  
CALL机  
CD机  
CD盒

采用人民日报语料库，分词训练集为 data/pku.training.utf8，如下所示：

pku\_trainning.utf8 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

迈向 充满 希望 的 新 世纪 —— 一九九八年 新年 讲话 （ 附 图片 1 张 ）

中共中央 总书记 、 国家 主席 江 泽民

（ 一九九七年 十二月 三十一日 ）

12月31日，中共中央总书记、国家主席江泽民发表1998年新年讲话《迈向充满希望的新世纪》

同胞们、朋友们、女士们、先生们：

在1998年来临之际，我十分高兴地通过中央人民广播电台、中国国际广播电台和中央电视台，

1997年，是中国发展历史上非常重要的很不平凡的一年。中国人民决心继承邓小平同志的遗

在这一年中，中国的改革开放和现代化建设继续向前迈进。国民经济保持了“高增长、低通胀”

在这一年中，中国的外交工作取得了重要成果。通过高层互访，中国与美国、俄罗斯、法国、

1998年，中国人民将满怀信心地开创新的业绩。尽管我们在经济社会发展中还面临不少困难，

实现祖国的完全统一，是海内外全体中国人的共同心愿。通过中葡双方的合作和努力，按照“一

台湾是中国领土不可分割的一部分。完成祖国统一，是大势所趋，民心所向。任何企图制造“两个

环顾全球，日益密切的世界经济联系，日新月异的科技进步，正在为各国经济的发展提供历史机遇

中国政府将继续坚持奉行独立自主的和平外交政策，在和平共处五项原则的基础上努力发展同世界

在这辞旧迎新的美好时刻，我祝大家新年快乐，家庭幸福！

谢谢！（新华社北京12月31日电）

在十五大精神指引下胜利前进——元旦献辞

我们即将以丰收的喜悦送走牛年，以昂扬的斗志迎来虎年。我们伟大祖国在新的一年里，将是

刚刚过去的一年，大气磅礴，波澜壮阔。在这一年，以江泽民同志为核心的党中央，继承邓小

1998年，是全面贯彻落实党的十五大提出的任务的第一年，各条战线改革和发展的任务都

今年是党的十一届三中全会召开20周年，是我们党和国家实现伟大的历史转折、进入改革开放

我们要更好地坚持解放思想、实事求是的思想路线。解放思想、实事求是，是邓小平理论的精髓。

我们要更好地坚持以经济建设为中心。各项工作必须以经济建设为中心，是邓小平理论的基本观

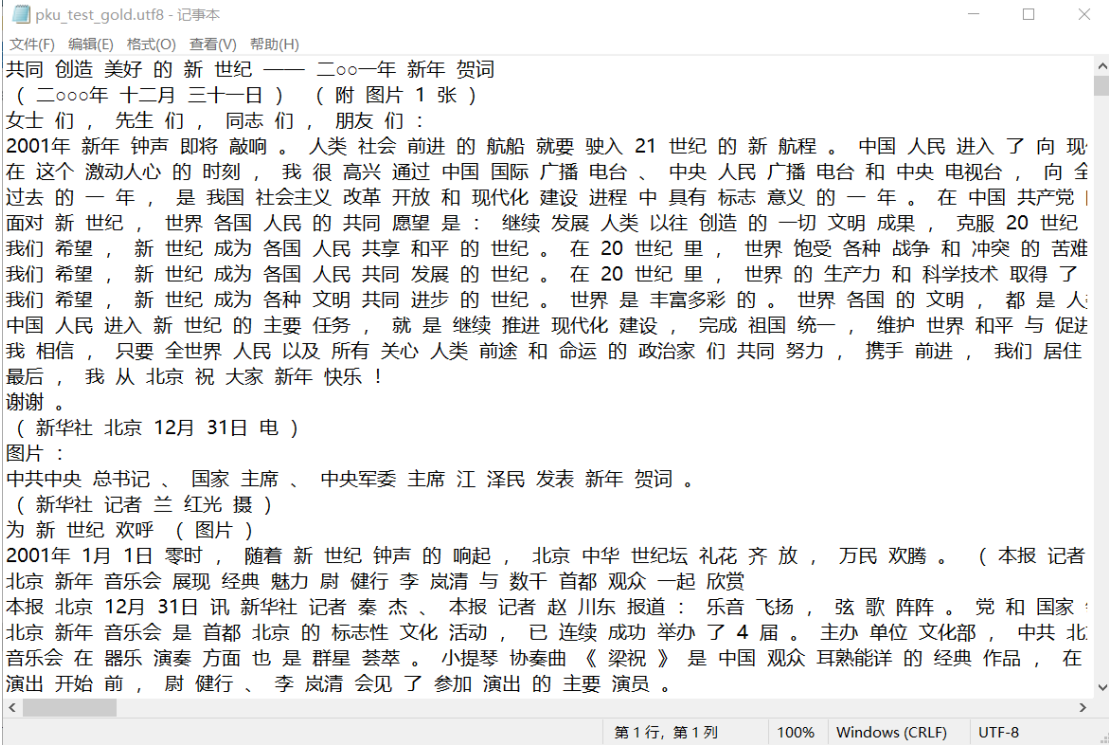
我们要更好地坚持“两手抓、两手都要硬”的方针。在坚持以经济建设为中心的同时，积极

我们要更好地发扬求真务实、密切联系群众的作风。这是把党的方针、政策落到实处，使改革

第1行, 第1列 100% Windows (CRLF) UTF-8

分词测试集为 `data/puk test.utf8`，如下所示：

分词测试集的结果为 data/puk\_test\_gold.uft8 为测试集的标准分词结果，如下所示：



pku\_test\_gold.uft8 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

共同 创造 美好 的 新 世纪 —— 二〇〇一年 新年 贺词  
( 二〇〇〇年 十二月 三十一日 ) ( 附 图片 1 张 )  
女士 们 , 先生 们 , 同志 们 , 朋友 们 :  
2001年 新年 钟声 即将 敲响 。 人类 社会 前进 的 航船 就要 驶入 21 世纪 的 新 航程 。 中国 人民 进入 了 向 现在 这个 激动人心 的 时刻 , 我 很 高兴 通过 中国 国际 广播 电台 、 中央 人民 广播 电台 和 中央 电视台 , 向 全 过去 的 一 年 , 是 我国 社会主义 改革 开放 和 现代化 建设 进程 中 具有 标志 意义 的 一 年 。 在 中国 共产党 面对 新 世纪 , 世界 各国 人民 的 共同 愿望 是 : 继续 发展 人类 以往 创造 的 一切 文明 成果 , 克服 20 世纪 我们 希望 , 新 世纪 成为 各国 人民 共享 和平 的 世纪 。 在 20 世纪 里 , 世界 饱受 各种 战争 和 冲突 的 苦难 我们 希望 , 新 世纪 成为 各国 人民 共同 发展 的 世纪 。 在 20 世纪 里 , 世界 的 生产 力 和 科学技术 取得 了 我们 希望 , 新 世纪 成为 各种 文明 共同 进步 的 世纪 。 世界 是 丰富多彩 的 。 世界 各国 的 文明 , 都 是 人 中国 人民 进入 新 世纪 的 主要 任务 , 就 是 继续 推进 现代化 建设 , 完成 祖国 统一 , 维护 世界 和平 与 促进 我 相信 , 只要 全世界 人民 以及 所有 关心 人类 前途 和 命运 的 政治家 们 共同 努力 , 携手 前进 , 我们 居住 最后 , 我 从 北京 祝 大家 新年 快乐 !  
谢谢 。  
( 新华社 北京 12月 31日 电 )  
图片 :  
中共中央 总书记 、 国家 主席 、 中央军委 主席 江 泽民 发表 新年 贺词 。  
( 新华社 记者 兰 红光 摄 )  
为 新 世纪 欢呼 ( 图片 )  
2001年 1月 1日 零时 , 随着 新 世纪 钟声 的 响起 , 北京 中华 世纪 坛 礼花 齐 放 , 万民 欢腾 。 ( 本 报 记者 北京 新年 音乐会 展现 经典 魅力 尉 健行 李 岚清 与 数千 首都 观众 一起 欣赏  
本 报 北京 12月 31日 讯 新华社 记者 秦 杰 、 本 报 记者 赵 川东 报道 : 乐音 飞扬 , 弦 歌 阵阵 。 党 和 国家 : 北京 新年 音乐会 是 首都 北京 的 标志性 文化 活动 , 已 连续 成功 举办 了 4 届 。 主办 单位 文化 部 , 中共 北 音乐会 在 器乐 演奏 方面 也 是 群星 荟萃 。 小提琴 协奏曲 《 梁祝 》 是 中国 观众 耳熟能详 的 经典 作品 , 在 演出 开始 前 , 尉 健行 、 李 岚清 会见 了 参加 演出 的 主要 演员 。

< 第 1 行, 第 1 列 100% Windows (CRLF) UTF-8

词性标注的测试集仍采用人民日报语料库，同时带有词性标记，经过处理过后，可以得到经过处理过后的语料库，取 70%作为训练集，取 30%作为测试集：



pku\_Tag.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

这些/r 名人/n 包括/v 宋代/t 名将/n 宗泽/n 、 /w 大/a 书法家/n 米芾/n 及/c 三国/t 时期/n 东吴/n 重臣/n 鲁肃/n 、 /w 中郎将/n 青岛市/n 楼山乡/n 成为/v 专利/n 乡/n  
本 报 /r 青岛/n 1月/t 10日/t 电/n 记者/n 宋学春/n 报道/v : /w 青岛市/n 李沧区/n 楼山乡/n 拥有/v 专利/n 和/c 专利/n 技术, 楼山乡/n 以/p 专利/n 为/v 突破口/n , /w 培育/v 全/a 乡/n 新/a 的/u 经济/n 增长点/n 。 /w 乡里/n 成立/v 了/u 乡/n 专利/n 街/n 广州/n 街头/s “ /w 走/v 鬼/a ” /w 多/a  
本 报 /r 记者/n 郑德刚/n  
新年/t 刚/d 过/v , /w 又/d 逢/v 农历/n 年关/t , /w 羊城/n 街面/n 上/f 到处/d 弥漫/v 着/u 新春/t 的/u 气息/n 。 /w 然而/c , 元旦/t 之后/f , /w 记者/n 连续/a 用/v 了/u 两/m 个/q 下午/t 的/u 时间/n , /w 在/p 广州/n 的/u 北京路/n 、 /w 沿江路/n 、 /w 1月/t 6日/t 临近/v 傍晚/t , /w 记者/n 又/d 来到/v 中山路/n 与/p 北京路/n 交叉/v 路口处/n 的/u 过街天桥/n 附近/f 。 /w 这 “ /w 都/d 有/v 什么/r 发票/n , /w 多少/r 钱/n — /m 张/q ? /w ” /w “ /w 你/r 想/v 要/v 什么样/r 的/u 发票/n ? /w 有/v ‘ /w 万/m ’ /w 字头/n 的/u 、 /w ‘ /w 千/m ’ /w 字头/n 的/u , /w 价 广州/n 究竟/d 有/v 多少/r 这样/r 的/u “ /w 走/v 鬼/a ” /w , /w 每天/r 又/d 有/v 多少/r 这种/r 似乎/d 一钱不值/i 却/d 又/d \* /w \* /w \* /w  
兜售/v 假/a 发票/n , /w 是/v 一/m 种/q 不法/b 行为/n , /w 理应/v 受到/v 严厉/a 打击/v 。 /w 但是/c , /w 在/p 有些/r 地方 贺卡/n : /w 太/d 俗/a 太/d 滥/a  
王群/n  
元旦/t 和/c 春节/t 之际/f , /w 我/r 收到/v 不少/m 贺卡/n , /w 但/c 没有/v 什么/r 兴奋/v 的/u 感觉/n , /w 反而/d 觉得/v 有 赞/v 民乐/n 走/v 进/v “ /w 金色/n 大厅/n ” /w  
李直/n  
据悉/v , /w 在/p 维也纳/n “ /w 金色/n 大厅/n ” /w , /w 将/d 上演/v 中国/n 民族/n 音乐/n 。 /w 由此/d 联想/v 到/v 在/p : 中华/n 文化/n 源远流长/i , /w 举世/n 公认/v 。 /w 但/c 由于/c 近代/t 中国/n 积弱积贫/l , /w 很多/m 中国/n 的/u 文化/n 艺 论/v 狼抓/v 落实/v  
本 报 /r 评论员/n  
做好/v 今年/t 各/r 方面/n 的/u 工作/v , /w 关键/n 是/v 要/v 狼抓/v 落实/v 。 /w  
狼抓/v 落实/v , /w 就/d 是/v 实事求是/i , /w 脚踏实地/i , /w 扎实/a 工作/v , /w 集中/v 精力/n 抓/v 大事/n , /w 注意/v 解

< 第 18 行, 第 36 列 100% Unix (LF) UTF-8

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
 迈向 充满 希望 的 新 世纪 —— 一九九八年 新年 讲话 （ 附 图片 1 张 ）  
 中共中央 总书记 、 国家 主席 江 泽民  
 （ 一九九七年 十二月 三十一日 ）  
 12月31日，中共中央总书记、国家主席江泽民发表1998年新年讲话《迈向充满希望的新世纪》  
 同胞们、朋友们、女士们、先生们：  
 在1998年来临之际，我十分高兴地通过中央人民广播电台、中国国际广播电台和中央电视台，  
 1997年，是中国发展历史上非常重要的很不平凡的一年。中国人民决心继承邓小平同志的遗  
 在这一年中，中国的改革开放和现代化建设继续向前迈进。国民经济保持了“高增长、低通胀”  
 在这一年中，中国的外交工作取得了重要成果。通过高层互访，中国与美国、俄罗斯、法国、  
 1998年，中国人民将满怀信心地开创新的业绩。尽管我们在经济社会发展中还面临不少困难，  
 实现祖国的完全统一，是海内外全体中国人的共同心愿。通过中葡双方的合作和努力，按照“一  
 台湾是中国领土不可分割的一部分。完成祖国统一，是大势所趋，民心所向。任何企图制造“两个  
 环顾全球，日益密切的世界经济联系，日新月异的科技进步，正在为各国经济的发展提供历史机遇  
 中国政府将继续坚持奉行独立自主的和平外交政策，在和平共处五项原则的基础上努力发展同世界  
 在这辞旧迎新的美好时刻，我祝大家新年快乐，家庭幸福！  
 谢谢！（新华社北京12月31日电）  
 在十五大精神指引下胜利前进——元旦献辞  
 我们即将以丰收的喜悦送走牛年，以昂扬的斗志迎来虎年。我们伟大祖国在新的一年里，将是  
 刚刚过去的一年，大气磅礴，波澜壮阔。在这一年，以江泽民同志为核心的党中央，继承邓小平  
 1998年，是全面贯彻落实党的十五大提出的任务的第一年，各条战线改革和发展的任务都  
 今年是党的十一届三中全会召开20周年，是我们党和国家实现伟大的历史转折、进入改革开放  
 我们要更好地坚持解放思想、实事求是的思想路线。解放思想、实事求是，是邓小平理论的精髓。实  
 我们要更好地坚持以经济建设为中心。各项工作必须以经济建设为中心，是邓小平理论的基本观  
 我们要更好地坚持“两手抓、两手都要硬”的方针。在坚持以经济建设为中心的同时，积极  
 我们要更好地发扬求真务实、密切联系群众的作风。这是把党的方针、政策落到实处，使改革和

### ① 程序结构:

```

1. class dict_tree:
2.     # 构建词典树
3.     def __init__(self):
4.         self.root = {} #词典树的根节点
5.         self.max_length = 0 # 最大长度
6.         self.endtag = '[end]' # 词典树的标记，有end 标记的才表
           示为一个词
7.
8.     def insert(self, word):
9.         node = self.root
10.        for chara in word:
11.            node = node.setdefault(chara, {}) #添加节点
12.            self.max_length = max(self.max_length, len(word))
13.            node[self.endtag] = '[end]'
14.
15.    def search(self, word):
16.        node = self.root

```

```

17.         for chara in word:
18.             if chara not in node:
19.                 return 0
20.             node = node[chara]
21.             if self.endtag in node:
22.                 return 1 #如果最后有 engtag 那么这个路径是词
23.             else:
24.                 return 0 #如果没有 那么在词典树中就没有找到该词

```

之后定义函数，来初始化词典树：

```

1. def init_dict_tree(dictfile_path):
2.     '''
3.     初始化词典树
4.     :param dictfile_path: 词典的路径
5.     :return: 形成的词典树
6.     '''
7.     dict = dict_tree()
8.     dictfile = open(dictfile_path, 'r' , encoding = "utf-
9.         8")
10.    for line in dictfile.readlines():
11.        dict.insert(line.strip())
12.    print("最长长度:{}".format(dict.max_length))
13.    return dict

```

之后定义函数，分别是正向匹配分词，逆向匹配分词以及双向匹配分词：

```

1. def forward_segment(dict: dict_tree, inputdata, outputdata)
2.     :
3.     '''
4.     正向匹配分词
5.     :param dict: 词典树
6.     :param inputdata: 输入文件路径
7.     :param outputdata: 要输出的文件路径
8.     :return: 返回一个列表 包含分词的结果 供双向分词使用
9.     '''
10.    inputfile = open(inputdata, "r", encoding="utf-8")
11.    outputfile = open(outputdata, "w", encoding="utf-8")
12.    result_list = []
13.    for line in inputfile.readlines():
14.        while len(line) > 0:
15.            max_len = dict.max_length
16.            if len(line) < max_len:
17.                max_len = len(line)
18.            #开始搜索
19.            word = line[0:max_len]
20.            while dict.search(word) == 0:

```

```

21.         if len(word) == 1:
22.             break
23.             #没有搜索到 减少一位 继续搜索
24.             word = word[0:(len(word)-1)]
25.         if word != '\n':
26.             outfile.write(word+" ")
27.         else:
28.             outfile.write(word)
29.             result_list.append(word)
30.             line = line[len(word):]
31.
32.     infile.close()
33.     outfile.close()
34.
35.     return result_list
36.
37. def backward_segment(dict: dict_tree, inputdata, outputdata
    ):
38.     '''
39.     逆向匹配分词
40.     :param dict: 词典树
41.     :param inputdata: 输入文件路径
42.     :param outputdata: 输出的文件路径
43.     :return: 返回一个列表 包含分词的结果 供双向分词使用
44.     '''
45.
46.     infile = open(inputdata, "r", encoding="utf-8")
47.     outfile = open(outputdata, "w", encoding="utf-8")
48.     result_list = []
49.     for line in infile.readlines():
50.         result = []
51.         while len(line) > 0:
52.             max_len = dict.max_length
53.             if len(line) < max_len:
54.                 max_len = len(line)
55.             #逆向查找 长度为 max_len 的词语
56.             word = line[(len(line) - max_len):]
57.             while dict.search(word) == 0:
58.                 if len(word) == 1:
59.                     break
60.                 # 减少一位
61.                 word = word[1:]
62.             result.append(word)
63.             line = line[0:(len(line) - len(word))]

```

```

64.         while len(result) > 0:
65.             res_word = result.pop()
66.             if res_word != '\n':
67.                 outfile.write(res_word+" ")
68.             else:
69.                 outfile.write(res_word)
70.                 result_list.append(res_word)
71.
72.     infile.close()
73.     outfile.close()
74.
75.     return result_list
76.
77. def count_single(wordlist : list):
78.     '''
79.     统计单字成词的个数
80.     :param wordlist: 分词列表
81.     :return: 单字成词的个数
82.     '''
83.     return sum(1 for word in wordlist if len(word) == 1)
84.
85. def bidirectional_segment(forward_list, backward_list, outp
    utdata):
86.     '''
87.     双向匹配分词
88.     词数更少优先级更高
89.     单字更少优先级更高
90.     都相等时逆向匹配优先级更高
91.     :param forward_list: 正相匹配分词列表
92.     :param backward_list: 逆向匹配分词列表
93.     :param outputdata: 输出文件路径
94.     :return:
95.     '''
96.
97.     outfile = open(outputdata, "w", encoding="utf-8")
98.     result_list = []
99.     if len(forward_list) < len(backward_list):#词数更少优先级
        更高
100.         result_list = forward_list
101.         for word in forward_list:
102.             if word != '\n':
103.                 outfile.write(word+" ")
104.             else:
105.                 outfile.write(word)

```



```

106.         elif len(forward_list) > len(backword_list):
107.             result_list = backword_list
108.             for word in backword_list:
109.                 if word != '\n':
110.                     outputfile.write(word + " ")
111.                 else:
112.                     outputfile.write(word)
113.             else:
114.                 if count_single(forward_list) < count_single(back
word_list):#单字更少优先级更高
115.                     result_list = forward_list
116.                     for word in forward_list:
117.                         if word != '\n':
118.                             outputfile.write(word + " ")
119.                         else:
120.                             outputfile.write(word)
121.                 else:#都相等时逆向匹配优先级更高
122.                     result_list = backword_list
123.                     for word in backword_list:
124.                         if word != '\n':
125.                             outputfile.write(word + " ")
126.                         else:
127.                             outputfile.write(word)
128.             outputfile.close()
129.             return result_list

```

速度评测函数如下 评价基于词典分词的效率:

```

1. def evaluate_speed(dict: dict_tree, inputdata, outputdata):
2.     '''
3.     测量分词的效率
4.     :param dict: 词典树
5.     :param inputdata: 输入文件路径
6.     :param outputdata: 输出文件的路径
7.     :return:
8.     '''
9.     start_time = time.time()
10.    num = 0
11.    with open(inputdata, 'r', encoding="utf-8") as fp:
12.        for line in fp.readlines():
13.            num = num + len(line)
14.
15.    # forward_segment(dict, inputdata, outputdata)
16.    backward_segment(dict, inputdata, outputdata)
17.
18.    elapsed_time = time.time() - start_time

```

```
19.     print("共{}字".format(num), end="")
20.     print("耗时{}s".format(elapsed_time))
```

统计分词的字数和消耗的时间。

## ② 算法设计:

词典树的构造算法设计:

通过 python 中的 `dict` 类型来构造出词典, 在向树中插入节点时, 通过使用 `dict` 类型的 `setdefault` 函数, 来添加没有出现的节点。同时, 在向词典树中插入节点是时, 要注意为每个词添加一个`[end]`标记, 标记每一个词, 当且仅当词典树的后面有`[end]`标记时, 从根节点到该节点可以成词。

在初始化词典树时, 读取词典文件 `dict.txt`, 然后调用 `dict_tree` 类中的插入函数, 将词典中的每一个词插入到词典树中。

正向匹配算法的算法设计:

为了减少算法的时间复杂性, 我们在 `dict_tree` 类中增加了一个变量 `max_length`, 表示所有录入的词典中的最长词的长度。在正向搜索时, 我们从第一个字开始, 以长度为 `max_length` 开始进行查找, 查找是否在词典树中。若不在词典树中, 那么长度减一, 再进行查找, 直到查找到或者长度变为 1 (单字成词的情况) 时停止。然后输出到 `output` 文件中。

逆向匹配算法的算法设计:

与正向匹配算法同理, 我们进行逆向搜索词, 同样是以长度为 `max_length` 开始再词典树中查找, 直到查找到或者长度变为 1 (单字成词的情况), 输出到 `output` 文件中。

双向匹配算法的算法设计:

双向匹配算法是正向匹配算法与逆向匹配算法的结合, 在得到正向匹配分词的结果列表和逆向匹配分词的结果列表后, 按照以下方式对结果进行选择:

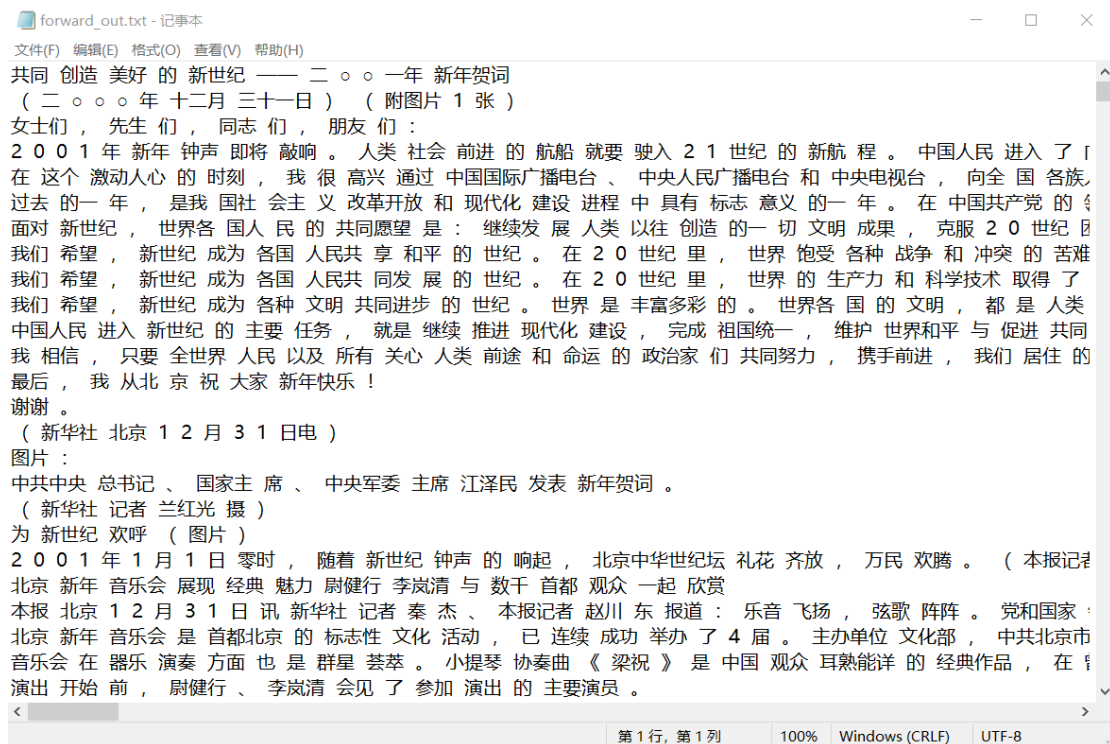
词数更少的分词语句 优先级更高

单字成词的数目更少的分词语句 优先级更高

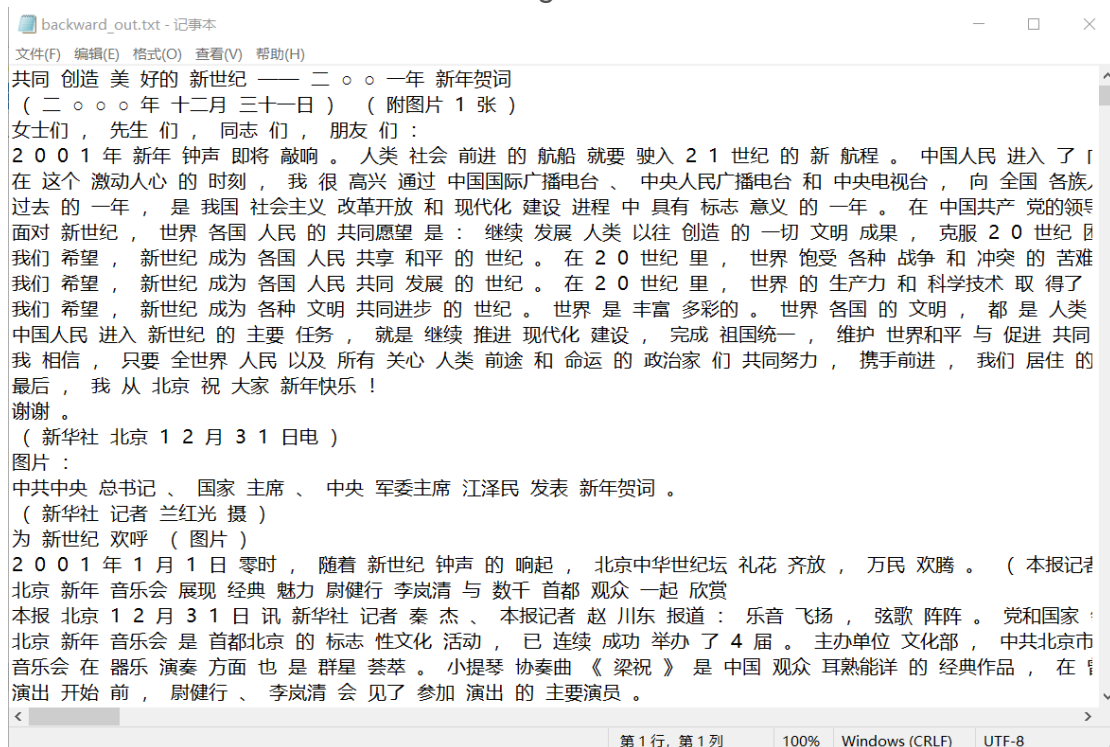
在其他情况都相等的情况下 优先选择逆向匹配算法

## ③程序运行的结果:

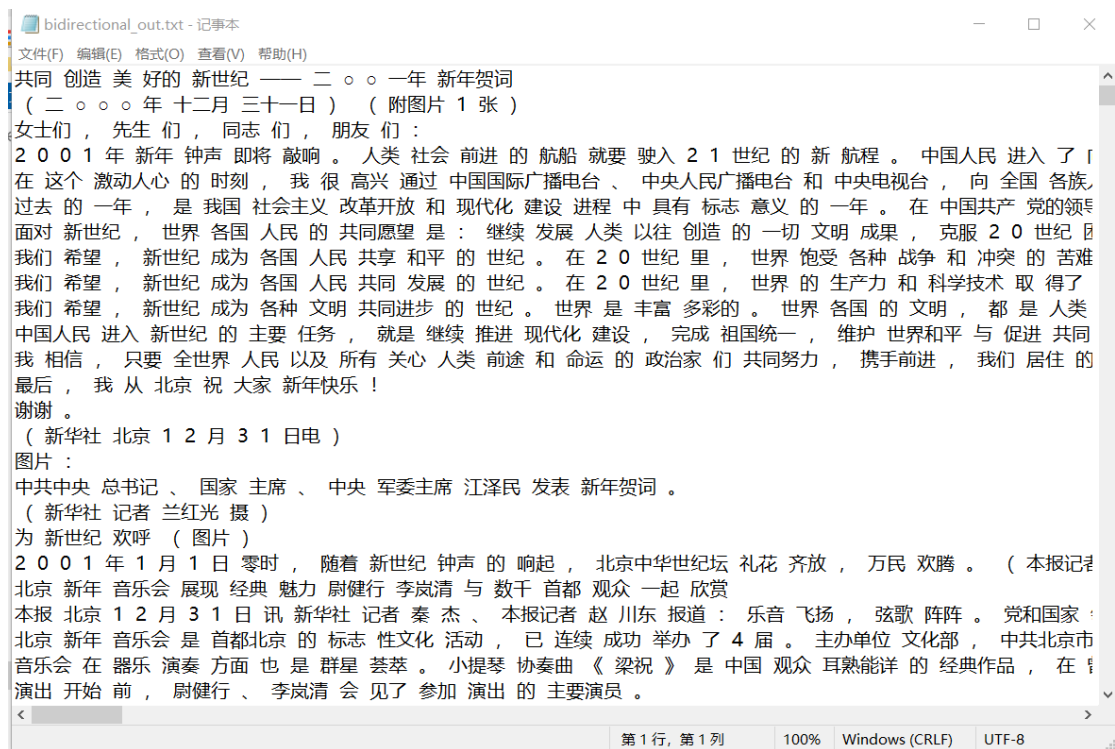
正向分词的运行结果位于 `data/seg_result/forward_out.txt`:



逆向分词的运行结果位于 data/seg\_result/backward\_out.txt



双向分词的运行结果位于 data/seg\_result/bidirectional\_out.txt



#### ④ 程序运行的评测结果：

结果如下表：

	效率(s/万字)	Precious	Recall	F-Score
正向匹配算法	12.35	80.35%	78.27%	79.30%
逆向匹配算法	13.54	80.66%	78.58%	79.61%
双向匹配算法	6.256	80.45%	78.73%	79.58%

#### ⑤基于词典分词过程中的问题以及解决方法：

**问题 1：**在开始做基于词典的分词时，我在初始化录入词典时，使用的是直接将词典中的词放入列表之中，使用 `list.contain()`函数来判断该词是否在列表之中。在运行正向匹配分词、逆向匹配分词和双向匹配分词时，运行时间过长，大约需要 6 个小时才能运行结束，算法的效率过低。

**解决方案：**改用词典树的方法，使用词典书的方法之后，效率提高程度极其明显，现在只需要 1-2 秒就可以运行结束。

### 3. 基于统计的分词方法：

#### ① 程序结构：采用隐马尔科夫模型（HMM）来进行中文统计分词

程序中包含一个类 `HMMSegment()`，类中包含以下函数：

类的初始化以及参数的初始化函数：

```

1.     def __init__(self):
2.         self.epsilon = sys.float_info.epsilon
3.         self.state_list = ['B', 'M', 'E', 'S']
4.         self.start_p = {} # 初始化概率矩阵
5.         self.trans_p = {} # 状态转移矩阵
6.         self.emit_p = {} # 发射矩阵
7.         self.state_dict = {} # 状态集合
8.         self.__init_parameters()
9.
10.    def __init_parameters(self):
11.        for state in self.state_list:
12.            self.start_p[state] = 1/len(self.state_list)
13.            self.trans_p[state] = {s: 1/len(self.state_list)
14.                ) for s in self.state_list}
15.            self.emit_p[state] = {}
16.            self.state_dict[state] = 0

```

统计序列标签函数：

```

1.     def __label(self, word):
2.         out = []
3.         if len(word) == 1:
4.             out = ['S']
5.         else:
6.             out += ['B'] + ['M'] * (len(word) - 2) + ['E']
7.         return out

```

隐马尔可夫模型的训练函数：

```

1. def train(self, dataset: list):
2.     '''
3.     训练，包含三个矩阵初始矩阵 发射矩阵 状态矩阵的填充
4.     :param dataset: 数据集
5.     :return:
6.     '''
7.     if not dataset or len(dataset) == 0:
8.         print('数据为空')
9.         return
10.
11.     line_nb = 0
12.     for line in dataset:
13.         line = line.strip()
14.         if not line:
15.             continue
16.         line_nb += 1
17.

```

```

18.         char_list = [c for c in line if c != ' ']
19.         word_list = line.split()
20.         state_list = []
21.         for word in word_list:
22.             state_list.extend(self.__label(word))
23.
24.         assert len(state_list) == len(char_list)
25.
26.         for index, state in enumerate(state_list):
27.             self.state_dict[state] += 1
28.
29.             if index == 0:
30.                 self.start_p[state] += 1
31.             else:
32.                 self.trans_p[state_list[index-
33. 1]][state] += 1
34.                 self.emit_p[state_list[index]][char_list[in
35. dex]] \
36.                     = self.emit_p[state_list[index]].get(ch
37. ar_list[index], 0) + 1
38.
39.         # 训练得到 初始概率矩阵
40.         self.start_p = {state: (num+self.epsilon)/line_nb f
41. or state, num in self.start_p.items()}
42.
43.         # 训练得到 转移概率矩阵
44.         self.trans_p = {
45.             pre_state:
46.                 {
47.                     cur_state: (cur_num+self.epsilon)/self.
48. state_dict[pre_state] for cur_state, cur_num in value.items
49. ()
50.                 } for pre_state, value in self.trans_p.item
51. s()
52.         }
53.
54.         # 训练得到 发射概率矩阵
55.         self.emit_p = {
56.             state:
57.                 {
58.                     char: (char_num+self.epsilon)/self.stat
59. e_dict[state] for char, char_num in value.items()
60.                 } for state, value in self.emit_p.items()
61.         }

```

```
54.         print('训练完成')
```

维特比算法函数，返回最大的概率及最佳路径：

```
1. def __viterbi(self, sentence: str):
2.     '''
3.
4.     :param sentence:需要划分的句子
5.     :return: 最优最大概率的状态序列
6.     '''
7.     dp = [{}]
8.     path = {}
9.     for state in self.state_list:
10.        dp[0][state] = self.start_p[state]*self.emit_p[
11.            state].get(sentence[0], self.epsilon)
12.        path[state] = [state]
13.    for index in range(1, len(sentence)):
14.        dp.append({})
15.        new_path = {}
16.
17.        for cur_state in self.state_list:
18.            emitp = self.emit_p[cur_state].get(sentence
19.                [index], self.epsilon)
20.            (prob, pre_state) = max(
21.                [(dp[index - 1][pre_state] * self.trans
22.                    _p[pre_state].get(cur_state, self.epsilon) * emitp, pre_sta
23.                    te)
24.                for pre_state in self.state_list if dp
25.                    [index - 1][pre_state] > 0]
26.            )
27.            dp[index][cur_state] = prob
28.            new_path[cur_state] = path[pre_state]+[cur_
29.                state]
30.        path = new_path
31.
32.        if self.emit_p['M'].get(sentence[-
33.            1], self.epsilon) > \
34.            self.emit_p['S'].get(sentence[-
35.                1], self.epsilon):
36.            (prob, state) = max([(dp[len(sentence)-
37.                1][state], state) for state in ('E', 'M')])
38.        else:
39.            (prob, state) = max([(dp[len(sentence)-
40.                1][state], state) for state in self.state_list])
```

```
32.  
33.         return prob, path[state]
```

## 分词函数:

```

1. def cut(self, sentence: str):
2.     '''
3.         切分句子 进行分词
4.         :param sentence: 句子
5.         :return: 分词的序列
6.     '''
7.     res = []
8.     if sentence is None or len(sentence) == 0:
9.         return res
10.
11.     #re 正则表达式区分汉语的序列 之后直接通过 re 来划分
12.     sentence
13.     re_han = re.compile("([\u4E00-\u9FD5a-zA-Z0-9+&\._%\-\-]+)", re.U)
14.     blocks = re_han.split(sentence)
15.     # print(blocks)
16.     for blk in blocks:
17.         if not blk:
18.             continue
19.         if re_han.match(blk):
20.             divide = []
21.             prob, pos_list = self.__viterbi(blk)
22.             begin_, next_ = 0, 0
23.             for i, char in enumerate(blk):
24.                 pos = pos_list[i]
25.                 if pos == 'B':
26.                     begin_ = i
27.                 elif pos == 'E':
28.                     divide.append(blk[begin_:i + 1])
29.                     next_ = i + 1
30.                 elif pos == 'S':
31.                     divide.append(char)
32.                     next_ = i + 1
33.             if next_ < len(blk):
34.                 divide.append(blk[next_:])
35.             #print(divide)
36.             for word in divide:
37.                 res.append(word)
38.         else:

```



```
39.                 res.append(blk)
40.         return res
```

## ② 算法设计:

### 隐马尔可夫模型:

采用隐马尔可夫模型，我们根据训练集，我们可以得到状态的概率函数。在隐马尔可夫模型中，状态的状态转换过程时隐蔽的，可观察事件的随机过程四隐蔽的观察状态转换过程的随机函数。

其中包含三个矩阵：初始状态概率矩阵 隐含状态概率矩阵 观测状态概率矩阵

中文分词中，HMM 模型有 4 个状态， **B M E S**， **B** 表示开始，**M** 表示词中，**E** 表示词尾，**S** 表示单个字符的情况。

初始概率矩阵表示的是序列头的状态分布，即分词中每个句子的开头标记为 **BMES** 的概率。

状态转移概率为状态序列内，不同状态之间转移的分布，状态转移概率构成了 **4\*4** 的状态转移矩阵，标记处 **4** 种标记之间转化的概率。

观测概率矩阵为由某个隐藏状态输出为某个观测状态的概率。

在训练的过程中，统计每个句子开头序列表及，最后初一句子的总个数，可以得到了初始概率矩阵

根据训练集，统计不同序列状态转化的个数，利用极大似然估计，将矩阵的每个元素除以语料中标记字的个数，得出转移概率的矩阵。

根据训练集，统计由隐藏状态输出为观测状态的个数，得到 **4\*N** 的矩阵，将矩阵的每个元素除以语料中标记的个数，得到感测概率矩阵。

### 维特比算法:

通过维特比算法求给定观测序列的 条件下，出现最佳序列路径以及该路径出现的概率，通过动态规划的算法来求解。由于受到观测序列的约束，起点给定后，递推，每次递推进入一个新的状态，长度增加，根据转移概率和发射概率计算划分，找出局部最优的路径。

之后找出最终时刻的最大概率，最后进行回溯，根据数组回溯到前驱状态。之后可以得出最优概率即状态序列。

## ③ 程序运行结果:

运行结果位于 `data/seg_result/Hmm_out.txt` 文件中，如图所示:



#### ④ 程序运行的评测结果：

	效率(万字/s)	Precious	Recall	F-Score
基于统计分词	2.145	78.43%	76.88%	77.65%

可以看出 HMM 单独分词的准确率和召回率和基于词典的分词的评测值差别不大。

#### ⑤基于统计的分词方法中出现的問題及解决方案：

**问题 1：**对于取一个极小概率，我们可以直接将他设置成 `float['INF']`，方便我们来进行统计。

在路径回溯的时候，可以通过确定边界条件，及对于每一个句子，最后一个字的状态只可能是 E 或者 S，不可能是 M 或者 B，所以可以通过这样排除一部分的路径回溯，进行剪枝。

**问题 2：**再对进行分词的概率是，我们可以通过数据平滑的方式，在本代码中使用的是+1 平滑。但是我们可以发现，在未进行数据平滑时，PRF 值与进行数据平滑后的 PRF 值差别不大，可见数据平滑对 HMM 分词的影响效果不大。

**问题 3：**经过训练得出来的模型，在进行测试时，如果测试集中有从未出现在训练集中的词，会发生 `IndexError` 现象，解决方案为直接通过该词得前后词性分析，由训练模型中的分词转移概率矩阵得到最大的概率，来判断如何进行分词。在词性标注中，也遇到了类似的问题。

## 4. 分词的评价方法:

### ①程序结构:

判断分词的评价方法函数:

```
1. def cal_fscore(testfile, goldfile):
2.     '''
3.
4.     :param testfile: 测试集的文件路径
5.     :param goldfile: 答案集的文件路径
6.     :return: 返回三个值 P R F1
7.     '''
8.     # calculate f-score 计算f-score
9.
10.    f_gold = open(goldfile, "r", encoding='UTF-8')
11.    f_test = open(testfile, "r", encoding='UTF-8')
12.
13.    total_gold_seg = 0
14.    for line in f_gold.readlines():
15.        total_gold_seg += len(line.split())
16.
17.    total_test_seg = 0
18.    for line in f_test.readlines():
19.        total_test_seg += len(line.split())
20.
21.    total_correct_seg = 0 #所有分词正确的数量
22.    total_error_seg = 0 #所有错误分词的数量
23.
24.    f_gold.seek(0)
25.    f_test.seek(0)
26.    for line_gold, line_test in zip(f_gold.readlines(), f_test.readlines()):
27.        dict_gold = {}
28.        dict_test = {}
29.        for words in line_gold.strip().split():
30.            if words in dict_gold:
31.                dict_gold[words] += 1
32.            else:
33.                dict_gold[words] = 1
34.        for words in line_test.strip().split():
35.            if words in dict_test:
36.                dict_test[words] += 1
37.            else:
38.                dict_test[words] = 1
```

```

39.
40.     for words in dict_test:
41.         if words in dict_gold:
42.             if dict_gold[words] >= dict_test[words]:
43.                 total_correct_seg += dict_test[words]
44.             else:
45.                 total_correct_seg += dict_gold[words]
46.                 total_error_seg += dict_test[words] - d
47.         else:
48.             total_error_seg += dict_test[words]
49.
50.
51. P_score = total_correct_seg / total_test_seg
52. R_score = total_correct_seg / total_gold_seg
53. F_score = 2 * P_score * R_score / (P_score + R_score)
54. print("Precious = ", P_score)
55. print("Recall = ", R_score)
56. print("F-Score = ", F_score)
57. f_test.close()
58. f_gold.close()

```

## ② 算法结构：

对于 P、R、F1 值的计算：

记标准答案的所有区间构成一个集合 A，记分词结果所有单词区间构成集合 B，那么，相应的 P、R、F1 的计算公式如下：

$$P = \frac{|A \cap B|}{|B|}$$

$$R = \frac{|A \cap B|}{|A|}$$

$$F1 = \frac{2 \times P \times R}{P + R}$$

所以，我们只需要统计总的测试集中的分词数目，统计总的目标集中的分词数目，之后，我们统计所有分词正确的词的数目，接可以计算出来 P、R 值，之后根据 P、R 值即可统计出 F1 值.

## ③运行结果：

	效率 (s/万字)	Precious	Recall	F-Score
正向匹配算法	12.35	80.35%	78.27%	79.30%
逆向匹配算法	13.54	80.66%	78.58%	79.61%
双向匹配算法	6.256	80.45%	78.73%	79.58%
基于统计分词	2.145	78.43%	76.88%	77.65%

将之前的表统计起来即可

## 5. 词性标注方法:

### ① 程序结构:

定义类 `class HmmPostag()`，用隐马尔可夫模型来做词性标注的类，包含有以下几个函数:

下面是初始化类的函数

```
1.     def __init__(self):
2.         self.trans_prop = {}# 转移概率矩阵
3.         self.emit_prop = {}# 发射概率矩阵
4.         self.start_prop = {}# 初始状态矩阵
5.         self.poslist = []
6.         self.trans_sum = {}
7.         self.emit_sum = {}
```

下面是更新转移概率矩阵的函数:

```
1.     def __upd_trans(self, curpos, nxtpos):
2.         """更新转移概率矩阵
3.         输入:
4.             curpos (string): 当前词性
5.             nxtpos (string): 下一词性
6.         """
7.         if curpos in self.trans_prop:
8.             if nxtpos in self.trans_prop[curpos]:
9.                 self.trans_prop[curpos][nxtpos] += 1
10.            else:
11.                self.trans_prop[curpos][nxtpos] = 1
12.        else:
13.            self.trans_prop[curpos] = {nxtpos: 1}
```

下面是更新发射概率矩阵的函数:

```
1.     def __upd_emit(self, pos, word):
2.         """更新发射概率矩阵
3.         输入:
4.             pos (string): 词性
5.             word (string): 词语
6.         """
7.         if pos in self.emit_prop:
8.             if word in self.emit_prop[pos]:
9.                 self.emit_prop[pos][word] += 1
10.            else:
11.                self.emit_prop[pos][word] = 1
12.        else:
13.            self.emit_prop[pos] = {word: 1}
```

下面是更新初始状态矩阵的函数:

```

1.     def __upd_start(self, pos):
2.         """更新初始状态矩阵
3.         输入:
4.             pos (string): 初始词语的词性
5.         """
6.         if pos in self.start_prop:
7.             self.start_prop[pos] += 1
8.         else:
9.             self.start_prop[pos] = 1

```

下面是训练 hmm 模型的函数，通过训练集球的转移矩阵、发射矩阵和初始状态矩阵：

```

1.     def train(self, data_path):
2.         """训练 hmm 模型、求得转移矩阵、发射矩阵、初始状态矩阵
3.         输入:
4.             data_path (string): 训练数据文件的地址
5.         """
6.         f = open(data_path, 'r', encoding='utf-8')
7.         for line in f.readlines():
8.             line = line.strip().split()
9.             # 统计初始状态的概率
10.            self.__upd_start(line[0].split('/')[1])
11.            # 统计转移概率、发射概率
12.            for i in range(len(line) - 1):
13.                self.__upd_emit(line[i].split('/')[1], line
14.                                [i].split('/')[0])
15.                self.__upd_trans(line[i].split('/')[1],
16.                                line[i + 1].split('/')[1])
17.            i = len(line) - 1
18.            self.__upd_emit(line[i].split('/')[1], line[i].
19.                            split('/')[0])
20.        f.close()
21.        # 记录所有的 pos
22.        self.poslist = list(self.emit_prop.keys())
23.        self.poslist.sort()
24.        # 统计 trans、emit 矩阵中各个 pos 的归一化分母
25.        num_trans = [
26.            sum(self.trans_prop[key].values()) for key in s
27.            elf.trans_prop
28.        ]
29.        self.trans_sum = dict(zip(self.trans_prop.keys(), n
30.                                um_trans))
31.        num_emit = [
32.            sum(self.emit_prop[key].values()) for key in se
33.            lf.emit_prop

```

```

29.         ]
30.         self.emit_sum = dict(zip(self.emit_prop.keys(), num
    _emit))

```

下面是通过维特比算法对已经分词好的结果进行词性标注预测，分词的要求为“A B C D”（每个词通过空格分开）

```

1. def predict(self, sentence):
2.     """Viterbi 算法预测词性
3.     输入：
4.         sentence (string): 分词后的句子（空格隔开）
5.     返回值
6.         list: 词性标注序列
7.     """
8.     sentence = sentence.strip().split()
9.     posnum = len(self.poslist)
10.    dp = pd.DataFrame(index=self.poslist)
11.    path = pd.DataFrame(index=self.poslist)
12.    # 初始化 dp 矩阵 (posnum * wordsnum 存储每个 word 每个 pos 的最大概率)
13.    start = []
14.    num_sentence = sum(self.start_prop.values()) + posnum
15.    for pos in self.poslist:
16.        sta_pos = self.start_prop.get(pos, 1e-16) / num_sentence
17.        sta_pos *= (self.emit_prop[pos].get(sentence[0], 1e-16) /
18.                    self.emit_sum[pos])
19.        sta_pos = math.log(sta_pos)
20.        start.append(sta_pos)
21.    dp[0] = start
22.    # 初始化 path 矩阵
23.    path[0] = ['_start_'] * posnum
24.    # 递推
25.    for t in range(1, len(sentence)): # 句子中第 t 个词
26.        prob_pos, path_point = [], []
27.        for i in self.poslist: # i 为当前词的 pos
28.            max_prob, last_point = float('-inf'), ''
29.            emit = math.log(self.emit_prop[i].get(sentence[t], 1e-16) / self.emit_sum[i])
30.            for j in self.poslist: # j 为上一次的 pos
31.                tmp = dp.loc[j, t - 1] + emit
32.                tmp += math.log(self.trans_prop[j].get(i, 1e-16) / self.trans_sum[j])
33.            if tmp > max_prob:

```

```

34.             max_prob, last_point = tmp, j
35.             prob_pos.append(max_prob)
36.             path_point.append(last_point)
37.             dp[t], path[t] = prob_pos, path_point
38.         # 回溯
39.         prob_list = list(dp[len(sentence) - 1])
40.         cur_pos = self.poslist[prob_list.index(max(prob_list))]
41.         path_que = []
42.         path_que.append(cur_pos)
43.         for i in range(len(sentence) - 1, 0, -1):
44.             cur_pos = path[i].loc[cur_pos]
45.             path_que.append(cur_pos)
46.         # 返回结果
47.         postag = []
48.         for i in range(len(sentence)):
49.             postag.append(sentence[i] + '/' + path_que[-
50.             i - 1])
50.         return postag

```

## ② 算法设计：

实际上，仍然采用的是 HMM 隐马尔可夫模型以及维特比算法来进行词性标注，与分词所采用的模型一样。

然而词性标注问题关键在于消除某些具有多种词性的歧义。比如观测，既可以作为动词，又可以作为名词，而词性标注的关键就是消除这样的歧义。

即我们先通过隐马尔可夫模型观察序列匹配最可能的系统确定隐藏的状态参数，训练生成形成模型参数。根据输入句子输出此行序列，所以我们就将词性标记序列作为隐藏状态，把句子中的单词作为观测序列。通过训练语料库，确定状态转移矩阵、初始概率矩阵、发射概率矩阵，确定了隐藏状态（标记集）以及多个观测符号（词性）。之后，我们仍然通过维特比算法计算出最优的转移序列以及转移的概率，即最优的词性标注的路径，通过维特比算法来解码。

我们将人民日报语料库以 9: 1 进行划分作为训练集和测试集，来进行测试和结果评测。

同时，为了方便进行测评，我们通过将人民日报语料库中筛出来的测试集，去掉词性标注，作为测试集，将没有去掉词性标注的部分作为测试集的目标集。之后在进行评测时，直接对比 HMM 词性标注分词的结果与人民日报语料库中的标准分词结果进行评测。

## ③ 运行结果：

词性标注的结果放在 data\postag\_result\pos\_ans.txt 中，结果如下所示：





```

6.     '''
7.     postaglist = postagstr.strip().split(" ")
8.     res = []
9.
10.    for word in postaglist:
11.        wordlist = word.split("/")
12.        pretag = wordlist[-1]
13.        if ']' in wordlist[-1]:
14.            pretaglist = wordlist[-1].split("]")
15.            pretag = pretaglist[0]
16.        res.append(pretag)
17.    # print(res)
18.    return res

```

第二个函数为 `cal_postag(testfile, goldfile)` 用于计算出分词的准确性，如下所示：

```

1. def cal_postag(testfile, goldfile):
2.     '''
3.     用于测量正确值
4.     :param testfile: 测试集的文件路径
5.     :param goldfile: 答案集的文件路径
6.     :return: 返回准确值
7.     '''
8.     # calculate f-score 计算f-score
9.     f_gold = open(goldfile, "r", encoding='UTF-8')
10.    f_test = open(testfile, "r", encoding='UTF-8')
11.    gold = f_gold.readlines()
12.    test = f_test.readlines()
13.
14.    total_correct = 0
15.    total_word = 0
16.
17.    for gold_line, test_line in zip(gold, test):
18.        gold_tag = get_tag(gold_line)
19.        test_tag = get_tag(test_line)
20.
21.        for res_gold, res_test in zip(gold_tag, test_tag):
22.            # print("{}=={}".format(res_gold, res_test))
23.            if res_gold == res_test:
24.                total_correct += 1
25.                total_word += 1
26.
27.    # print(total_word)
28.    # print(total_correct)
29.    print(total_correct/total_word)

```

## ② 算法描述:

即提取出来测试集和训练集中每一行的词性, 然后对比训练集和测试集中的词性, 如果相同, 那么 `total_correct+1` , 之后计算 所有正确词性标注的词数/总的词性标注的次数。

如同中文分词一样, 词性标注的评测方法相同。在词性标注时, 我使用的是人民日报语料库的标准分词结果, 所以测试集中标签总数等于预测的的标签总数, 所以 `P`、`R`、`F1` 均相等, 退化为准准确率, 即:

$$Accuracy = \frac{\text{预测正确的标签总数}}{\text{标签总数}}$$

即选择 `Accuracy` 作为评价指标。

## ③ 运行结果:

运行结果如下图所示, 与 `HMM` 词性标注的测试结果相同:

	效率 (词/s)	准确率 Accuracy
HMM 词性标注	22.85	92.54%

# 四、命名实体识别实验过程:

## 1. 命名实体识别方法描述:

### ① 程序结构:

`LSTM.py` 文件, 其中包含是的 `LSTM` 的模型类, 以及 `LSTM` 类属性值以及方法:

其中包含有 `tag_label` 即识别人名、地名、组织实体所对应的标签值

类的初始化方法 `init`

以及 `LSTM` 模型的正向方法 `forward`

以及得出结果函数 `test`

```
1. '''
2. LSTM 模型类
3. 通过 torch.nn 来训练网络
4. '''
5. import torch
6. import torch.nn as nn
7. from torch.nn.utils.rnn import pad_packed_sequence, pack_padded_sequence
8.
9.
10. class BiLSTM(nn.Module):
11.     tag_label = {"O": 0,
12.                  "B-PER": 1, "I-PER": 2,
13.                  "B-LOC": 3, "I-LOC": 4,
```

```

14.         "B-ORG": 5, "I-ORG": 6
15.     }
16.
17.     def __init__(self, vocab_size, emb_size, hidden_size, out_size):
18.         """初始化参数:
19.             vocab_size:字典的大小
20.             emb_size:词向量的维数
21.             hidden_size: 隐向量的维数
22.             out_size:标注的种类
23.         """
24.         super(BiLSTM, self).__init__()
25.         self.embedding = nn.Embedding(vocab_size, emb_size)
26.         self.bilstm = nn.LSTM(emb_size, hidden_size,
27.                                batch_first=True,
28.                                bidirectional=True)
29.
30.         self.lin = nn.Linear(2*hidden_size, out_size)
31.
32.     def forward(self, sents_tensor, lengths):
33.         emb = self.embedding(sents_tensor) # [B, L, emb_size]
34.
35.         packed = pack_padded_sequence(emb, lengths, batch_first=True)
36.         rnn_out, _ = self.bilstm(packed)
37.         # rnn_out:[B, L, hidden_size*2]
38.         rnn_out, _ = pad_packed_sequence(rnn_out, batch_first=True)
39.
40.         scores = self.lin(rnn_out) # [B, L, out_size]
41.
42.         return scores
43.
44.     def test(self, sents_tensor, lengths):
45.         logits = self.forward(sents_tensor, lengths) # [B, L, out_size]
46.         batch_tagids = torch.max(logits, dim=2)[1]
47.
48.         return batch_tagids

```

`data.py` 包含的是数据读取的文件

其中只包含有一个函数 `build_corpus`，即读取文件中的数据

```

1. '''
2. 用作读取数据

```

```

3. '''
4.
5. def build_corpus(filepath):
6.     """读取数据"""
7.
8.     word_lists = [] # 词列表
9.     tag_lists = [] # 状态列表
10.    with open(filepath, 'r', encoding='utf-8') as f:
11.        word_list = []
12.        tag_list = []
13.        for line in f:
14.            if line != '\n':
15.                word, tag = line.strip('\n').split()
16.                word_list.append(word)
17.                tag_list.append(tag)
18.            else:
19.                word_lists.append(word_list)
20.                tag_lists.append(tag_list)
21.                word_list = []
22.                tag_list = []
23.
24.    return word_lists, tag_lists

```

`main.py` 是程序的主函数，包含有数据读取、模型的训练模块，模型的载入与保存，将测试结果写入文件中的功能。

```

1. from data import build_corpus
2. from LSTM import BiLSTM
3. import pickle
4.
5. def extend_maps(word2id, tag2id):
6.     word2id['<unk>'] = len(word2id)
7.     word2id['<pad>'] = len(word2id)
8.     tag2id['<unk>'] = len(tag2id)
9.     tag2id['<pad>'] = len(tag2id)
10.    return word2id, tag2id
11.
12. def save_model(model, file_name):
13.     """用于保存模型"""
14.     with open(file_name, "wb") as f:
15.         pickle.dump(model, f)
16.
17. def load_model(file_name):
18.     """用于加载模型"""
19.     with open(file_name, "rb") as f:

```

```

20.         model = pickle.load(f)
21.     return model
22.
23. def bilstm_train(train_word_lists, train_tag_lists, dev_word_lists, dev_tag_lists, test_word_lists, test_tag_lists, word2id, tag2id):
24.     vocab_size = len(word2id)
25.     out_size = len(tag2id)
26.     bilstm_model = BiLSTM(vocab_size, 100, 300, out_size)
27.     bilstm_model.train(train_word_lists, train_tag_lists,
28.                        dev_word_lists, dev_tag_lists, word2id, tag2id)
29.     save_model(bilstm_model, "bilstm.pkl")
30.
31.     pred_tag_lists, test_tag_lists = bilstm_model.test(
32.         test_word_lists, test_tag_lists, word2id, tag2id)
33.
34.     return pred_tag_lists
35.
36. def main():
37.     # 第一步 读取数据:
38.     train_word_lists, train_tag_lists, word2id, tag2id = build_corpus("train.txt")
39.     dev_word_lists, dev_tag_lists = build_corpus("dev.txt")
40.     test_word_lists, test_tag_lists = build_corpus("test.txt")
41.
42.     # 第二步 训练Bi-LSTM:
43.     #LSTM 模型训练需要再word2id tag2id 中增加标志<pad> <unk>
44.     bilstm_word2id, bilstm_tag2id = extend_maps(word2id, tag2id)
45.
46.
47.     lstm_pred = bilstm_train(
48.         train_word_lists, train_tag_lists,
49.         dev_word_lists, dev_tag_lists,
50.         test_word_lists, test_tag_lists,
51.         bilstm_word2id, bilstm_tag2id,
52.     )
53.     print(lstm_pred)
54.
55.     #第三步 载入模型进行识别:
56.     bilstm_word2id, bilstm_tag2id = extend_maps(word2id, tag2id)

```

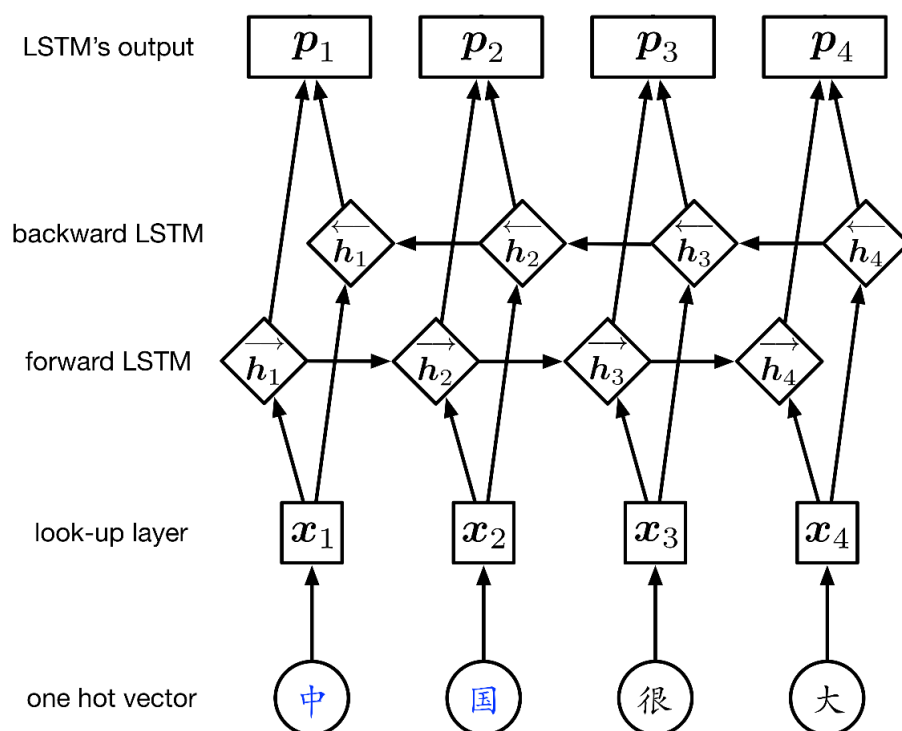
```

57.     bilstm_model = load_model("bilstm.pkl")
58.     target_tag_list = bilstm_model.test(test_word_lists, te
        st_tag_lists,
59.                                         bilstm_w
        ord2id, bilstm_tag2id)
60.     # print(target_tag_list)
61.
62.     #第四步 将数据写入测试结果文件中
63.     tag_label = {"O": 0,
64.                  "B-PER": 1, "I-PER": 2,
65.                  "B-LOC": 3, "I-LOC": 4,
66.                  "B-ORG": 5, "I-ORG": 6
67.                  }
68.     tag_label_list = [i for i,j in tag_label.items()]
69.     fw = open("ans.txt", 'w', encoding = "utf-8")
70.     for word, taglist in zip(test_word_lists, target_tag_li
        st):
71.         if taglist in tag_label:
72.             fw.write(word + "\t" + tag_label_list[taglist]
        + "\n")
73.
74.     fw.close()
75.
76. if __name__ == "__main__":
77.     main()

```

## ②算法设计:

本算法通过 biLSTM 网络来求解，LSTM 网络的模式如下图所示：



LSTM 是一种特殊的 RNN 类型，用于学习长期以来信息。LSTM 是依靠神经网络超强的非线性拟合能力，在训练时将样本通过高维空间中的复杂非线性变换，学习到从样本到标注的函数，之后使用这个函数为指定的样本预测每个 token 的标注。我们使用 `torch.nn.LSTM()` 来进行 LSTM 模型的构造。其中包含，`input_size` 为输入到 LSTM 单元的维度，`hidden_size` 确定隐含层的维度，之后经过的 `Layers` 为叠加的层数。

如图，先通过单热点将词转化成向量，通过词向量作为 Look-up Layer，输入到前向 LSTM，经过 LSTM 的 forward 层之后得到 LSTM 输出。经过 forward 层之后在经过一次反向传播层，双向 LSTM 能更好的捕捉序列之间的依赖关系。根据神经网络学习的标注的函数，得到每个词向量的结果，根据代码中的值，主要为 BI 标注。得到的值为 0 时，标注为 O；得到的值为 1 时，标注为 B-PER，2 为 I-PER，3 为 B-LOC，4 为 I-LOC，5 为 B-ORG，6 为 I-ORG。之后根据训练出的函数，得到一定的值，根据不同的值得到不同的标注。

LSTM 模型对于序列标注的结果的有点在于简单粗暴，不需要做繁杂的特征工程，直接训练即可，实际操作性较为简单。

### ③程序运行结果：

运行结果位于 `NER/ans.txt` 中，如下图所示：

其中，中共中央 被标为组织实体，是正确的，

但是，中国致公党十一大 也被标记为一个组织实体，不完全是正确的。





中	B-ORG
共	I-ORG
中	I-ORG
央	I-ORG
致	O
中	B-ORG
国	I-ORG
致	I-ORG
公	I-ORG
党	I-ORG
十	I-ORG
一	I-ORG
大	I-ORG
的	O
贺	O
词	O
各	O
位	O
代	O
表	O
、	O
各	O
位	O
同	O

由于时间原因以及测试集太少，准确率的评测不能说明问题，所以命名实体识别的结果不做结果评测。

## 五、程序使用说明：

### 1. 基于词典分词的程序使用说明：

基于词典分词的程序使用说明位于 Dict\_Tree\_Segment.py 中：

① 当想要进行三种分词都使用时，调用 main1() 函数即可

② 当只调用其中其中一种分词时，在 main 函数中之后直接调用函数，词典已经在 main() 函数中初始化好：

```
if __name__ == "__main__":  
    dict = init_dict_tree("data/dict.txt")
```

forward\_segment(dict: dict\_tree, inputdata, outputdata) 以及

```
backward_segment(dict: dict_tree, inputdata, outputdata)
```

中的 inputdata outputdata 分别是测试文件的地址以及输出文件的地址

③ 当调用速度测评函数时，调用 main2() 即可，需要测评哪个方法的速度，就在函数中调用哪个分词测评的函数即可。

## 2. 基于统计分词的程序使用说明：

基于词典分词的程序使用说明位于 Dict\_Tree\_Segment.py 中：

当想要使用 HMM 分词时，直接调用 main1() 函数即可，训练集的路径为 train\_file，结果函数的路径为 fw。

当使用 HMM 速度评测时，直接调用 main2() 函数即可。

## 3. 分词评测方案的程序使用说明：

分词评测函数位于 calculate\_division.py 文件中，直接调用 cal\_fscore() 即可，第一个参数为自己分词的结果的文件路径，第二个参数为标准的分词结果的文件路径。

## 4. 词性标注程序使用说明：

词性标注程序位于 PosTag\_word.py 文件中，直接运行即可，hmm.train() 中的参数是训练集的文件路径，fp 是测试集的文件指针，fw 是结果存放的文件指针。

该程序运行过后会自动显示出运行时长和分词的总次数。

## 5. 词性标注测评方案的程序使用说明：

词性标注测评方案程序位于 calculate\_postag.py 中，直接运行 cal\_postag()，第一个参数为自己运行的词性标注结果的文件路径，第二个参数为标准的词性标注的文件路径。

## 6. 命名实体识别的程序使用说明：

直接调用 main() 函数即可，其中，如果文件位置找不到，调整 main() 函数的第一步读取文件数据，修改其中的文件路径即可。