词法分析

# HanLP词法分析器概览

|  |  |  |  |
| --- | --- | --- | --- |
| 任务 | 算法 | 原理 | 特点 |
| 分词 | **最短路径分词** | Viterbi | 速度和精度的最佳平衡 |
| **N-最短路径分词** | topN最短路径 |
| 感知机分词 | 字构词 | 精度、新词发现 |
| CRF分词 |
| **极速词典分词** | 词典分词 | 速度 |
| 词性标注 | **HMM词性标注** | HMM | 速度 |
| 感知机词性标注 | 感知机 | 精度 |
| CRF词性标注 | CRF |
| 实体识别 | 中国人名识别 | HMM-Viterbi角色标注 | 速度 |
| 音译人名识别 | 层叠隐马 |
| 日本人名识别 |
| 地名识别 | HMM-Viterbi角色标注 |
| 机构名识别 | 层叠隐马 |
| 感知机命名实体识别 | 感知机 | 精度 |
| CRF命名实体识别 | CRF |

# 最短路径分词（viterbi）

基本思想：首先根据词典，找出字串中所有可能的词（也称全切分），然后构造词语切分有向无环图（也称作粗分词图或粗分词网）。每个词对应图中的一条有向边。若赋给相应的边长一个权值（该权值可以是常数，也可以是所构成的词的属性值），然后根据该切分图，在起点到终点的所有路径中，求出长度值（包括权值）为最短的一条路径，这条路径上包含的词就是该句子的切分结果。若每个结点处记录N个最短路径值，则该方法也称**N-最短路径算法**。

为进一步提高切分精度，在词典中增加词的属性值，即给每个词也给权重。这样每个词在汉字串中的权重不同（即构成的有向图的边不为等长）。最简单的词的权重可以用词频表示，高频词的权重大，低频词的权重小。具体的权重值可以通过大规模语料库获得。

虽然HanLP中提供了dijkstra算法的实现，但是当前HanLP中最短路径分词使用的是viterbi算法。

例子：他说的确实在理

**粗分词图的生成**



词图生成的HanLP代码为<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/WordBasedSegment.java> generateWordNet(WordNet)

**遍历计算过程和回溯分词过程**



1. node列与to列

node列的词语为粗分词网中所有的词，to列为在node列为词word\_node的情况下，后边接的所有可能的词word\_to。第1个词语前边有一个“始”词，最后一个词语后边有一个“末”词。

1. begin2node\_w的计算

表示从“始”到node词的最短路径权值。可以从待计算值所在行的node列读取出word词，在to列中以待计算值所在行开始向上查找word，找到word所在行后（以首次遇到的词为准），begin2to\_w列所对应的值就是待计算值。见图中下划线。第一个词对“始-他”的begin2node\_w的值为0。

1. node2to\_w的计算

由node+w构成的2gram串的概率，也就是转移概率，计算公式为

其中，为平滑参数值取0.1，为词语node的训练集频次值，为训练集中词形总数，为词语node和词语w的共现频次值，为平滑因子。

计算的HanLP代码为<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/utility/MathUtility.java> calculateWeight(Vertex from, Vertex to)。“始”的频次取为，“始-他”的共现频次值为“他”作为句首的频次，“理-末”的共现频次值为“理”作为句末的频次。

1. begin2to\_w\_n的计算

表示从“始”到to词的最短路径权值。begin2to\_w\_n = begin2node\_w + node2to\_w。

1. begin2to\_w\_o

表示记录在to词下的，到to词的最短路径权值，它的初始值为0，之后由begin2to\_w来更新。

1. from

表示词语to的前驱词。

，

可以看表中(7,9),(8,10),(11,13),(12,14),(15,16),(17,18)成对行来验证该公式，其中只有(17.18)行满足了第3个式子。

1. begin2to\_w的计算

表示从“始”到to词的最短路径权值。

可以看表中(7,9),(8,10),(11,13),(12,14),(15,16),(17,18)成对行来验证该公式，其中只有(17,18)行满足了第3个式子。

（6）和（7）的HanLP实现代码<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/common/Vertex.java> updateFrom(Vertex from)

1. 回溯确定分词路径

从“末”开始向前回溯，末->理->在->确实->的->说->他，可以看表中黄色单元格进行验证。

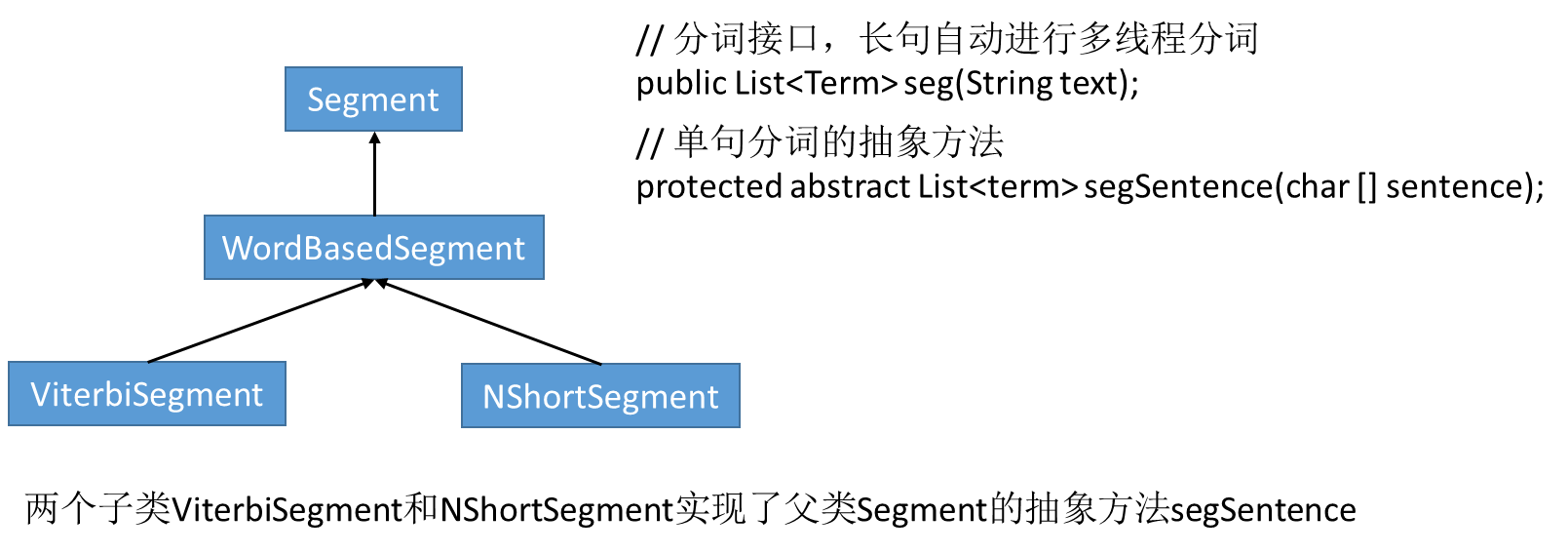
经过(6)、（7）两步，可以确保粗分词网中任意词的前驱都是最短路径的。

遍历计算过程和回溯过程的HanLP代码<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/Viterbi/ViterbiSegment.java> viterbi(WordNet wordNet)

|  |
| --- |
| private static List<Vertex> viterbi(WordNet wordNet)  {  // 避免生成对象，优化速度  LinkedList<Vertex> nodes[] = wordNet.getVertexes();  LinkedList<Vertex> vertexList = new LinkedList<Vertex>();  for (Vertex node : nodes[1])  {  node.updateFrom(nodes[0].getFirst());  }  for (int i = 1; i < nodes.length - 1; ++i)  {  LinkedList<Vertex> nodeArray = nodes[i];  if (nodeArray == null) continue;  for (Vertex node : nodeArray)  {  if (node.from == null) continue;  for (Vertex to : nodes[i + node.realWord.length()])  {  System.out.print(node.realWord + "\t" + to.realWord + "\t");  to.updateFrom(node);  }  }  }  Vertex from = nodes[nodes.length - 1].getFirst();  while (from != null)  {  vertexList.addFirst(from);  from = from.from;  }  return vertexList;  } |

# N-最短路径分词

前边已经提到，在最短路径分词中，若每个结点处记录N个最短路径值，则该方法称为**N-最短路径算法**。在HanLP中通过两个类ViterbiSegment和NshortSegment分别实现了最短路径分词和N-最短路径分词。首先看看两个类的关系。



这里长句自动按字数切成小句，小句之间再使用多线程进行分词。

我们看看两个类ViterbiSegment和NshortSegment的实现过程对比

|  |  |
| --- | --- |
| ViterbiSegment | NshortSegment |
| 构造词网结构 | |
| 生成一元词网，调用viterbi方法得到粗分结果 | 调用biSegment方法得到粗分结果 |
| 命名实体识别（根据配置ner） | |
| 数字识别（根据配置numberQuantifierRecognize） | |
| 索引分词（根据配置indexMode） | |
| 自定义词典干预（根据配置useCustomDictionary） | |
| 词性标注（根据配置speechTagging） | |

注意：

1. NshortSegment得到的时topN种分词结果，每种分词结果中有可能存在路径值相等的多个分词结果。
2. NshortSegment对得到的topN种分词结果并没有做筛选而只是返回了top1的一个分词结果。
3. 调用biSegment得到粗分的topN种分词结果后，会对每种分词结果中的每个结果做命名实体识别，之后只对最优的一个分词结果做数字识别、索引分词、自定义词典干预、词性标注。

有关于viterbi方法已经在最短路径分词中加以介绍，这里我们介绍biSegment方法。首先介绍biSegment整体流程。

1. 生成词网
2. 构建词图，也就是将每个可能词语的权重计算出来

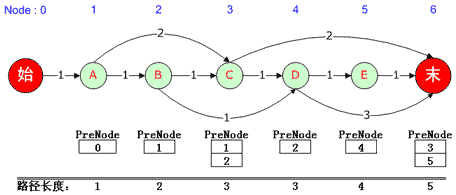
|  |
| --- |
| 打印词图：========按终点打印========  to: 1, from: 0, weight:04.09, word:始##始@他  to: 2, from: 1, weight:02.50, word:他@说  to: 3, from: 2, weight:03.24, word:说@的  to: 4, from: 2, weight:08.46, word:说@的确  to: 5, from: 3, weight:05.68, word:的@确  to: 6, from: 3, weight:05.67, word:的@确实  to: 7, from: 4, weight:11.23, word:的确@实  to: 7, from: 5, weight:11.46, word:确@实  to: 8, from: 4, weight:11.23, word:的确@实在  to: 8, from: 5, weight:11.46, word:确@实在  to: 9, from: 6, weight:03.17, word:确实@在  to: 9, from: 7, weight:11.21, word:实@在  to: 10, from: 6, weight:10.87, word:确实@在理  to: 10, from: 7, weight:11.21, word:实@在理  to: 11, from: 8, weight:11.18, word:实在@理  to: 11, from: 9, weight:07.00, word:在@理  to: 12, from: 10, weight:11.61, word:在理@末##末  to: 12, from: 11, weight:05.59, word:理@末##末 |

词图中每种词的weight值就是上节中node2to\_w的值。

1. 构建求N-最短路径实例，并返回N-最短路径
2. 计算出所有结点上可能的路径，为路径数据提供数据准备
3. 从短到长获取至多n条路经
4. 进行日期、数字合并

根据节点下标数组解释出对应的路径

下面详细介绍求N-最短路径的过程，首先看一张路径图示例



N-ShortPath的基本思想是：首先根据词典构建上边的词图。对每个节点找到它的所有前驱节点，由前驱节点from到当前节点to的权值计算出从“始“经前驱节点到当前节点的路径权值，按照路径权值大小对前驱节点排序，如果是1-NshortPath则保留top1个前驱节点，当然如果有并列的多个top1前驱节点则保留多个。如果是2-NshortPath则保留top2个前驱节点。最后在节点下边记录从“始”到“to”的top-n个路径长度。如果是1-NshrotPath则记录1个路径长度，如果是2-NshortPath则记录2个路径长度。我们以图中4号节点为例说明计算过程。4号节点的前驱节点有3号节点和2号节点，路径长度分别为4和3，于是对于1-NshortPath来说保留2号前驱节点，并在4号节点下记录路径长度为3。

我们来看看代码的实现

1. 首先初始化NshortPath类的成员喽，这里挑两个重要的成员说。

第一个成员是fromArray用来记录节点的前驱，对应路径图示例的每个“PreNode”。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | … | nValueKind |
| 1 | CQueue实例 | CQueue实例 | CQueue实例 | CQueue实例 |
| 2 | CQueue实例 | CQueue实例 | CQueue实例 | CQueue实例 |
| … | … | … | … | … |
| vertexCount | CQueue实例 | CQueue实例 | CQueue实例 | CQueue实例 |

其中，行表示节点，行数为节点数量；列表示最优长度值topN。

第二个成员是weightArray用来记录到每个节点的前N个最短路径上的花费之和，对应路径图示例的最后一行“路径长度”。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | … | nValueKind |
| 1 | double类型数 | double类型数 | double类型数 | double类型数 |
| 2 | double类型数 | double类型数 | double类型数 | double类型数 |
| … | … | … | … | … |
| vertexCount | double类型数 | double类型数 | double类型数 | double类型数 |

1. 查找到达词图中每种节点的所有前驱，并计算从“始”到经每个前驱节点到达该节点的权值（最短路径长度），每个节点只保留topN种个前驱节点。

实现方法为

<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/NShort/Path/NShortPath.java>

private void calculate(Graph inGraph, int nValueKind)

针对例子得到的结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 到达节点 | | top1前驱节点 | | top2前驱节点 | |
| 节点字 | to | from | weight | from | weight |
| 他 | 1 | 0 | 4.090693 |  |  |
| 说 | 2 | 1 | 6.586326 |  |  |
| 的 | 3 | 2 | 9.830333 |  |  |
| 的确 | 4 | 2 | 15.04746 |  |  |
| 确 | 5 | 3 | 15.50843 |  |  |
| 确实 | 6 | 3 | 15.49957 |  |  |
| 实 | 7 | 4 | 26.27918 | 5 | 26.97188 |
| 实在 | 8 | 4 | 26.27918 | 5 | 26.97188 |
| 在 | 9 | 6 | 18.67432 | 7 | 37.49335 |
| 在理 | 10 | 6 | 26.36763 | 7 | 37.49335 |
| 理 | 11 | 9 | 25.67691 | 8 | 37.46142 |
| 末 | 12 | 11 | 31.26441 | 10 | 37.97844 |

1. 获取topN最短路径（当存在并列路径值情况时，路径数可以大于topN）

实现方法为

<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/NShort/Path/NShortPath.java>

public List<int[]> getNPaths(int n)

针对例子得到的结果

粗分结果[他/rr, 说/v, 的/ude1, 确实/ad, 在/p, 理/n]

粗分结果[他/rr, 说/v, 的/ude1, 确实/ad, 在理/a]

1. 日期、数字合并

实现方法为

<https://github.com/hankcs/HanLP/blob/master/src/main/java/com/hankcs/hanlp/seg/NShort/Path/NShortPath.java>

protected static void generateWord(List<Vertex> linkedArray, WordNet wordNetOptimum)

至此，N-最短路径的基本方法介绍完毕，那么它与最短路径有什么区别呢？

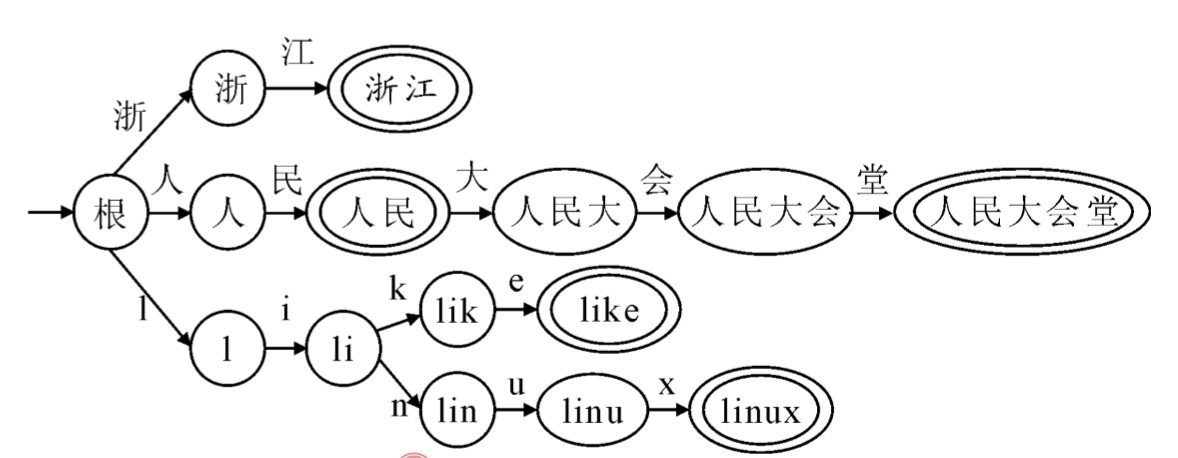
1. 第1个区别是节点上保留的最优路径前驱节点数。具体来说，当某个节点存在两个以上前驱时，N-最短路径一定会保留topN种路径值的所有前驱节点，而最短路径只会保留一个最短路径值的前驱节点。
2. HanLP在实现上为N-最短路径增加了数字、日期合并规则。
3. HanLP的N-最短路径方法最终返回的还是一个最优路径，并未对topN个分词结果做筛选策略，虽然对存在多个前驱的节点处保留了多个候选前驱，但是个人感觉两者相差应该不多，可能对歧义处有效果，但是对未登录词应该作用不大。说白了它也还是基于词典中单个词语的概率做的，其他的文本信息都没有用到。

# 极速词典分词

## 双数组Trie树

* 1. Trie树

Trie树又称单词查找树或键树，可以建立有效的数据检索组织结构，是一种高效的索引技术，适合字符串快速查找以及前缀匹配，因此Trie树常常用来构建中文分词使用。Trie树本质上是一个确定的有限状态自动机（DFA），每个结点代表自动机的一个状态，根据变量的不同进行状态转移，当到达结束状态或者无法转移的时候，完成查询。Trie树的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销，以达到提高效率的目的。Trie树的查询速率为o(1)到o(n)，n为查询串的长度。Trie树的结构如图所示



* 1. 双数组Trie树

双数组Trie树用两个数组来构造Trie树结构实现查找。两个数组的作用列于下边的两个表。

base数组

索引用字（词）编码值表示

|  |  |
| --- | --- |
| base[i]=0 | 表示该字（词）后边必须接其他字才能构成完整词 |
| base[i]<0 且 base[i] != -i | 表示从根节点到该字（词）的路径构成一个完整词，同时该字（词）后接其他字还可以构成其他完整词 |
| base[i]<0 且 base[i] == -i | 表示从根节点到该字（词）的路径构成一个完整词，同时该字（词）后边不会再有其他字（词）构成完整词 |

check数组

索引用字（词）编码值表示，数组元素为该字（词）对应的前边字（词）的编码值。

## Aho-CoraSick算法

1. Aho-CoraSick算法简称AC算法，通过将模式串预处理为确定有限状态自动机，扫描文本一遍就能结束。复杂度为o(n)，即与模式串的数量和长度有关。
2. 确定有限自动机。定义为一个五元组，其中是输入符号的有穷集合，是状态的有限集合，是初始状态，。是终止状态的集合，，是到（下一个状态）的映射，它支配着有限状态控制的行为，有时也称为状态转移函数。
3. Aho的匹配过程的三种情况以及三张表

|  |  |  |
| --- | --- | --- |
| goto表 | 表示从一个状态转移到另一个状态 |  |
| output表 | 输出一个词语集合 |  |
| failure表 | 无法转移到下一个状态，则按照failure表往回跳转 |  |

至此，可以看到Aho自动机比Trie树多一个failue表而已。

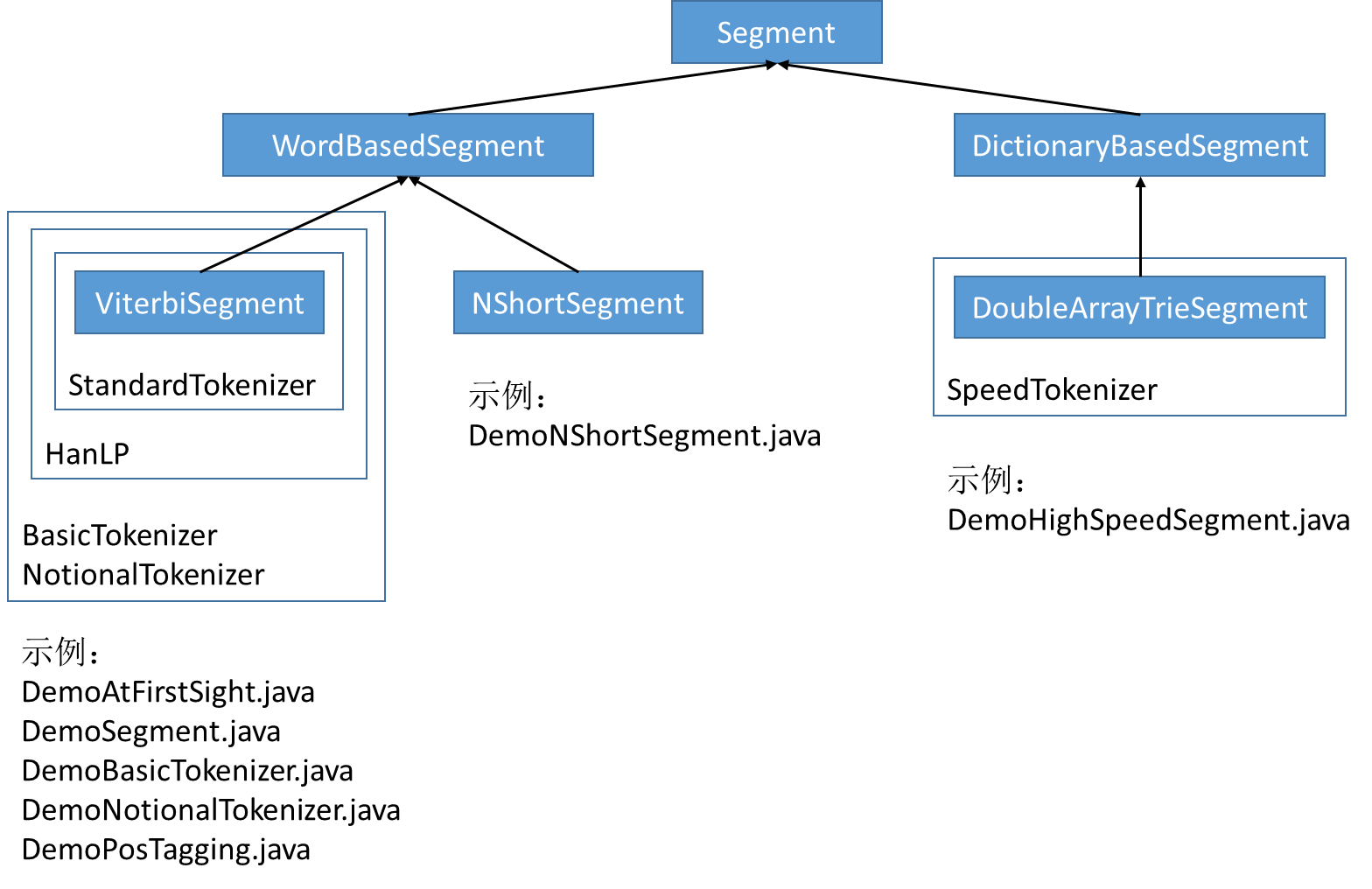
## AhoCorasickDoubleArrayTrie算法

1. 构建Trie树
2. 构建双数组Trie树
3. 构建fail表

## SpeedTokenizer类

HanLP中的核心词典已经替换为由AhoCorasickDoubleArrayTrie提供支持，HanLP中还实现了一个基于AhoCorasickDoubleArrayTrie的最长分词器。

至此，总结词法分析类。



这里说明一下，自动机的原理是实现词语的匹配，双数组Trie树用来表达自动机的结构，匹配的过程依靠的三张表。这种匹配是如何应用到最大匹配分词的还需要看DoubleArrayTrieSegment类中的segSentence方法的具体实现。由于正向最大匹配既不能很好解决分词歧义也无法解决未登录词识别，因此我没有详细阅读代码。

# HMM词性标注

在HanLP中所有继承WordBasedSegment的具体分词类的词性标注方法都是HMM的方法。在HanLP中对于未登录词的词性都标注为nz（其他专有名词）<https://github.com/hankcs/HanLP/issues/429>。HanLP中词性标记集见<http://www.hankcs.com/nlp/part-of-speech-tagging.html#h2-8>

## 隐马尔科夫模型

隐马尔可夫模型（Hidden Markov Model，HMM）是可用于标注问题的统计学习模型，描述由隐藏的马尔可夫链随机生成观测序列的过程，属于生成模型（联合概率分布）。

1. 隐马尔可夫模型的定义

隐马尔可夫模型是关于时序的概率模型，描述由一个隐藏的马尔可夫链随机生成不可观测的状态随机序列，再由各个状态生成一个观测而产生观测随机序列的过程。在词性标注任务中，观测序列指的是看到的词语序列，状态序列指的是词性标记序列。

1. 隐马尔可夫模型的表示

隐马尔可夫模型可以用三元符号表示，即，A，B，称为隐马尔可夫模型的三要素。有关于A，B，pi的具体介绍在例题中。

* 状态转移概率矩阵A与初始状态概率向量pi确定了隐藏的马尔可夫链，生成不可观测的状态序列。
* 观测概率矩阵B确定了如何从状态生成观测，与状态序列综合确定了如何产生观测序列。

1. 隐马尔可夫模型的两个基本假设

* 基本假设1：**齐次一阶**马尔可夫性假设。即假设隐藏的马尔可夫链在任意时刻t的状态只依赖于其前一时刻的状态，与其他时刻的状态及观测无关，也与时刻t无关。
* 基本假设2：观测独立性假设。即假设任意时刻的观测只依赖于该时刻的马尔可夫链的状态，与其他观测及状态无关。

两个基本假设也就是HMM的缺陷，在词性标注问题中，一阶性假设认为文本中位置t处的词性只依赖于它前边位置的词性，但实际文本中可能会由前边多个词的词性共同决定。而观测独立性假设认为词语之间没有关联，显然词语在不同的上下文中所具有的意义是不同的进而对外表现的词性也就不同，忽略词语之间的关联性显然也是不合适的。

1. 隐马尔可夫模型的三个基本问题

由于对于词性标注问题我们主要关心的是预测问题，因此这里只介绍预测问题。

第三个基本问题，预测问题，也称解码问题。已知模型和观测序列，求对给定观测序列条件概率P(I|O)最大的状态序列。即给定观测序列，求最有可能的对应的状态序列。

例：用HanLP对生的人民日报语料进行分词和词性标注，针对词性标注任务，统计出隐马尔科夫模型训练所需的2个矩阵A、B以及1个向量PI。

（1）初始状态概率向量PI的统计

首先按照HanLP词性标注集中所列的各种词性统计他们在语料中出现的次数，再按照下式计算各词性的初始概率。

PI = [pi\_1, pi\_2, …,pi\_n]，其中pi\_i = 词性i出现次数/词语总数

（2）观测概率矩阵B的统计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 词语1 | 词语2 | … | 词语m |
| 词性1 | b\_11 | b\_12 | … | b\_1m |
| 词性2 | b\_21 | b\_22 | … | b\_2m |
| … | … | … | … | … |
| 词性n | b\_n1 | b\_n2 | … | b\_nm |

矩阵的行表示词性标记，列表示词语，行数等于词性标记集中的标记总数，列数等于词汇表长度。元素b\_ij表示词性i发射出词语j的概率。它的计算公式为b\_ij = 词性i发射为词语j的次数/词性i发射出的词语总数

（3）状态转移概率矩阵A的统计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 词性1 | 词性2 | … | 词性n |
| 词性1 | a\_11 | a\_12 | … | a\_1n |
| 词性2 | a\_21 | a\_22 | … | a\_2n |
| … | … | … | … | … |
| 词性n | a\_n1 | a\_n2 | … | a\_nn |

矩阵的行和列都表示词性标记，行数和列数都等于词性标记集中的标记总数

# HanLP-停用词表的使用

## 停用词表的修改

停用词表在“pyhanlp\static\data\dictionary”路径下的“stopwords.txt”文件中，CoreStopWordDictionary.apply方法支持去除停用词。如果需要修改停用词表，则直接编辑文件“stopwords.txt”，之后删除路径下的“stopwords.txt.bin”，运行CoreStopWordDictionary.apply后即可自动生效。有关验证的方法见“验证是否生效”小节。

## 自定义词语过滤方法

用户可以通过编写“pyhanlp\static”路径下的“MyFilter.java”文件设置自己的词语过滤方法。应当注意这里处理的语言单位是词语，而不是字。编辑完毕后需要编译该文件并生成字节码文件，之后运行CoreStopWordDictionary.apply方法时就会自动调用用户自己的词语过滤方法了。这里给出一个自定义过滤方法的编写示例代码。

|  |
| --- |
| import os  from pyhanlp.static import STATIC\_ROOT, HANLP\_JAR\_PATH  java\_code\_path = os.path.join(STATIC\_ROOT, 'MyFilter.java')  with open(java\_code\_path, 'w') as out:  java\_code = """  import com.hankcs.hanlp.dictionary.stopword.CoreStopWordDictionary;  import com.hankcs.hanlp.dictionary.stopword.Filter;  import com.hankcs.hanlp.seg.common.Term;  public class MyFilter implements Filter  {  public boolean shouldInclude(Term term)  {  if (term.nature.startsWith('m')) return false; // 数词过滤  if (term.nature.startsWith('q')) return false; // 量词过滤  if (term.nature.startsWith('t')) return false; // 时间词过滤  if (term.nature.startsWith("w")) return false; // 过滤标点符号  return !CoreStopWordDictionary.contains(term.word); // 停用词过滤  }  }  """  out.write(java\_code)  os.system('javac -cp {} {} -d {}'.format(HANLP\_JAR\_PATH, java\_code\_path, STATIC\_ROOT)) |

## 验证是否生效

本节给出停用词表修改后以及使用了自定义词语过滤方法的示例代码。

|  |
| --- |
| from pyhanlp import \*  # 加载停用词类  CoreStopWordDictionary = JClass("com.hankcs.hanlp.dictionary.stopword.CoreStopWordDictionary")  # 加载自定义词语过滤逻辑  MyFilter = JClass('MyFilter')  CoreStopWordDictionary.FILTER = MyFilter()  term\_list = HanLP.segment(text)  CoreStopWordDictionary.apply(term\_list) |

# HanLP-自定义词表的使用

## 7.1 自定义词表的修改

自定义词表在“pyhanlp\static\data\dictionary\custom”路径下的“CustomDictionary.txt”文件中，HanLP.segment支持自定义此词表。如果需要修改自定义词表，则直接编辑文件“CustomDictionary.txt”，之后删除路径下的“CustomDictionary.txt.bin”，运行分词代码后即可自动生效。

## 7.2 验证是否生效

|  |
| --- |
| from pyhanlp import \*  term\_list = HanLP.segment(text) |