

武 汉 纺 织 大 学

《机器学习》课程报告

题目： 基于 HMM 的分词器及其应用

成 绩: _____

学 号: 1704210616

姓 名: 周佳爽

班 级: 计科 11706

指导教师: 杜小勤

报告日期: 2019 年 06 月 27 日

1 相关背景

1.1 机器学习现状

人工智能概念诞生于 1956 年，在半个多世纪的发展历程中，由于受到智能算法、计算速度、存储水平等多方面因素的影响，人工智能技术和应用发展经历了多次高潮和低谷。2006 年以来，以深度学习为代表的机器学习算法在机器视觉和语音识别等领域取得了极大的成功，识别准确性大幅提升，使人工智能再次受到学术界和产业界的广泛关注。[8] 在几十年的发展过程中人工智能发展出了机器学习、知识图谱、自然语言处理、人机交互、机器视觉、生物特征识别、智能语音等众多学科分支，其相关的重点技术如下：

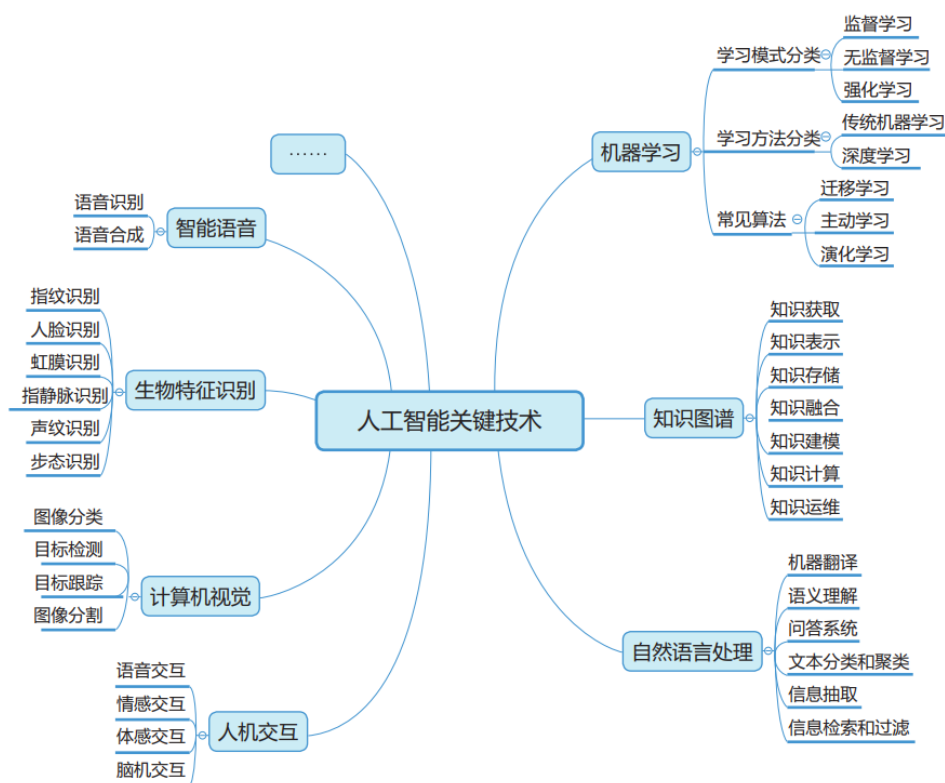


图 1: 人工智能重要技术分支 [11]

如图 (1), 本学期学习的机器学习就是人工智能关键技术的一个重要的分支, 那什么是机器学习呢? 机器学习 (Machine Learning) 是一门涉及统计学、系统辨识、逼近理论、神经网络、优化理论、计算机科学、

脑科学等诸多领域的交叉学科,研究计算机怎样模拟或实现人类的学习行为,以获取新的知识或技能,重新组织已有的知识结构使之不断改善自身的性能,是人工智能技术的核心。基于数据的机器学习是现代智能技术中的重要方法之一,研究从观测数据(样本)出发寻找规律,利用这些规律对未来数据或无法观测的数据进行预测。根据学习模式、学习方法以及算法的不同,机器学习存在不同的分类方法。[8].

譬如,优化算法一直是机器学习领域的重点,如何处理各种凸优化和非凸优化问题、如何处理分布式优化,避免局部最优解,一直是学者最关注的问题之一。其他值得关注的领域,还包括强化学习(reinforcement learning)、概率图模型(probabilistic graphical models)、统计关系学习(statistical relational learning)等。[13] 这些问题在未来都是会被重点研究.

根据学习模式可将其划分为监督学习、无监督学习、和强化学习。根据学习方法可将其划分为传统机器学习和深度学习。此外,机器学习的常见算法还包括迁移学习、主动学习和演化学习等。

本学期以李航的统计学习方法为蓝本的情况下,了解了监督学习中的感知机、K 邻近、朴素贝叶斯、逻辑斯蒂回归、HMM、CRF 等十大经典算法。在学习理论与实验中完成了各大经典算法的验证,以及初步的实验扩展,在此基础上确保了利用这十大经典算法是能够解决一些实际问题。

1.2 自然语言处理发展现况

从 20 世纪 40 年代到 50 年代末这个时期是自然语言处理的萌芽期,1948 年, C.E.Shannon 把离散马尔可夫过程的概率模型应用于描述语言的自动机。1956 年, 美国语言学家 N.Chomsky 从 Shannon 的工作中吸取了有限状态马尔可夫过程的思想, 首先把有限状态自动机作为一种工具来刻画语言的语法, 并且把有限状态语言定义为由有限状态语法生成的语言。这些早期的研究工作产生了“形式语言理论”(formal language theory) 这样的研究领域, 采用代数和集合论把形式语言定义为符号的序列。[9]

20 世纪 60 年代中期到 80 年代末期是自然语言处理的发展期。在自然语言处理的发展期,各个相关学科的彼此协作,联合攻关,取得了一些令人振奋的成绩。B.Vauquois 教授明确地提出,一个完整的机器翻译过程可以分为如下六个步骤:

1. 原语词法分析
2. 原语句法分析
3. 原语译语词汇转换
4. 原语译语结构转换
5. 译语句法生成
6. 译语词法生成

这六个步骤形成了“机器翻译金字塔”(MT pyramid, 图 (2))。其中第一、第二步只与原语有关,第五、第六步只与译语有关,只有第三、第四步牵涉到原语和译语二者。这就是机器翻译中的“独立分析—独立生成—相关转换”的方法。他们用这种方法研制的俄法机器翻译系统,已经接近实用水平。

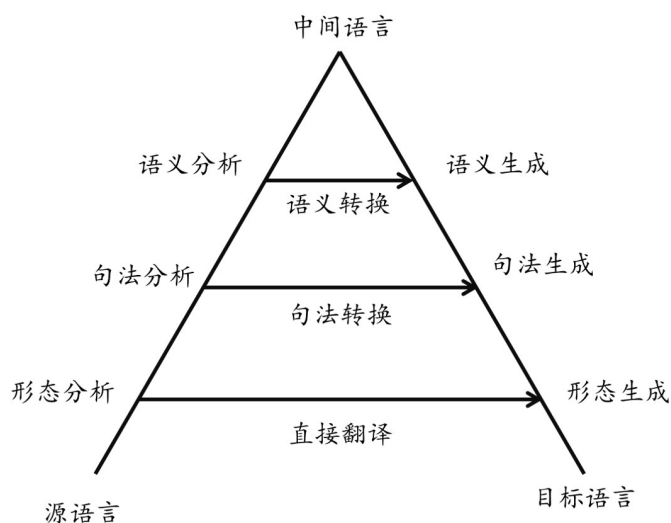


图 2: 机器翻译金字塔

从 20 世纪 90 年代开始,自然语言处理进入了繁荣期。自 1989 年以来,机器翻译的发展进入了一个新纪元。这个新纪元的重要标志是在

基于规则的技术中引入了语料库方法,其中包括统计方法、基于实例的方法、通过语料加工手段使语料库转化为语言知识库的方法等等。这种建立在大规模真实文本处理基础上的机器翻译,是机器翻译研究史上的一场革命,它将会把自然语言处理推向一个崭新的阶段。

21 世纪以来,由于国际互联网的普及,自然语言的计算机处理成了从互联网上获取知识的重要手段。与此同时,国内的自然语言处理研究有如下四个显著的特点:

- 基于句法—语义规则的理性主义方法受到质疑。随着语料库建设和语料库语言学的崛起,大规模真实文本的处理成为自然语言处理的主要战略目标
- 自然语言处理中越来越多地使用机器自动学习的方法来获取语言知识
- 自然语言处理中越来越重视词汇的作用,出现了强烈的“词汇主义”的倾向
- 统计数学方法越来越受到重视

1.3 课题的意义

在此我将结合所学习的知识完成一个基于 HMM 的中文分词器,并且结合朴素贝叶斯方法实现一个邮件分类器。完成本实验的意义有三:

就学习而言,在本学期的学习的算法, HMM 属于一个偏难的算法,在一周的时间学习理论知识加上实验过程时间不足,没能完全消化吸收相关知识,我希望通过本次的实验进一步的加深对 HMM 的理论了解与具体实现。在人工智能白皮书中将机器学习与自然语言处理划分为了两个不同的分支,但是随着时间的推移、技术的进步,自然语言处理中越来越多地使用机器自动学习的方法来获取语言知识,这也就是技术的必然性。通过这个实验能够使我在学习了机器学习的模型上应用到自然语言处理中,实现知识的迁移,初步的了解自然语言处理这个重要分支。

就技术而言, HMM 是一个基础的概率图模型 (probabilistic graphical models), 学习该算法属于承上启下的作用。隐马尔可夫模型 (HMM)

是语音识别的支柱模型，高斯混合模型 (GMM) 及其变种 K-means 是数据聚类的最基本模型，条件随机场 (CRF) 广泛应用于自然语言处理 (如词性标注，命名实体识别)，Ising 模型获得过诺贝尔奖，话题模型在工业界大量使用 (如腾讯的推荐系统)。等等。在熟练掌握了基础的 HMM 模型后对于学习其他的模型也会有非常好的帮助。

就未来研究方向而言，本科学习的知识始终是有限的，除去一些计算机的专业基础知识之外了解当前最新的发展方向，最新技术的机会少之又少，通过这次的短暂的与自然语言处理的接触，能够为未来工作或者学习研究选择方向提供一定的基础。

2 原理与算法描述

2.1 文本挖掘简介

2.1.1 分词的基本原理

在做文本挖掘的时候，首先要做的预处理就是分词。英文单词天然有空格隔开容易按照空格分词，但是也有时候需要把多个单词作为一个分词，比如一些名词如 “New York”，需要作为一个词看待。而中文由于没有空格，分词就是一个需要专门去解决的问题了。无论是英文还是中文，分词的原理都是类似的，本文就对文本挖掘时的分词原理做一个总结。

现代分词都是基于统计的分词，而统计的样本内容来自于一些标准的语料库。假如有一个句子：“小明来到荔湾区”，我们期望语料库统计后分词的结果是：“小明/来到/荔湾/区”，而不是“小明/来到/荔/湾区”。那么如何做到这一点呢？从统计的角度，我们期望“小明/来到/荔湾/区”这个分词后句子出现的概率要比“小明/来到/荔/湾区”大。如

果用数学的语言来说，如果有一个句子 S，它有 m 种分词选项如下：

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n_1} \\ A_{21} & A_{22} & \dots & A_{2n_2} \\ \vdots & & & \\ A_{m1} & A_{m2} & \dots & A_{mn_m} \end{bmatrix} \quad (1)$$

其中下标 n_i 代表第 i 种分词的词个数。如果我们从中选择了最优的第 r 种分词方法，那么这种分词方法对应的统计分布概率应该最大，即：

$$r = \underbrace{\operatorname{argmax}}_i P(A_{i1}, A_{i2}, \dots, A_{in_i}) \quad (2)$$

但是我们的概率分布 $P(A_{i1}, A_{i2}, \dots, A_{in_i})$ 并不好求出来，因为它涉及到 n_i 个分词的联合分布。在自然语言处理中，为了简化计算，我们通常使用马尔科夫假设，即每一个分词出现的概率仅仅和前一个分词有关，即：

$$P(A_{ij}|A_{i1}, A_{i2}, \dots, A_{i(j-1)}) = P(A_{ij}|A_{i(j-1)}) \quad (3)$$

于是 (2) 就能够表示为下式：

$$P(A_{i1}, A_{i2}, \dots, A_{in_i}) = P(A_{i2}|A_{i1})P(A_{i3}|A_{i2})\dots P(A_{in_i}|A_{i(n_i-1)}) \quad (4)$$

而通过我们的标准语料库，我们可以近似的计算出所有的分词之间的二元条件概率，比如任意两个词 w_1, w_2 ，它们的条件概率分布可以近似的表示为：

$$P(w_1|w_2) = \frac{P(w_1, w_2)}{P(w_2)} \approx \frac{\operatorname{freq}(w_1, w_2)}{\operatorname{freq}(w_2)}$$

$$P(w_2|w_1) = \frac{P(w_1, w_2)}{P(w_1)} \approx \frac{\operatorname{freq}(w_1, w_2)}{\operatorname{freq}(w_1)}$$

其中 $\operatorname{freq}(w_1, w_2)$ 表示 w_1, w_2 在语料库中相邻一起出现的次数，而其中 $\operatorname{freq}(w_1), \operatorname{freq}(w_2)$ 分别表示 w_1, w_2 在语料库中出现的统计次数。利用语料库建立的统计概率，对于一个新的句子，我们就可以通过计算

各种分词方法对应的联合分布概率，找到最大概率对应的分词方法，即为最优分词。

2.1.2 N 元模型

当然只依赖于前一个词太武断了，我们能不能依赖于前两个词呢？即：

$$P(A_{i1}, A_{i2}, \dots, A_{in_i}) = P(A_{i1})P(A_{i2}|A_{i1}) * P(A_{i3}|A_{i2}, A_{i1},) \dots P(A_{in_i}|A_{i(n_i-2)}, A_{i(n_i-1)}, \dots A_{i1}) \quad (5)$$

这样也是可以的，只不过这样联合分布的计算量就大大增加了。我们一般称只依赖于前一个词的模型为二元模型 (Bi-Gram model)，而依赖于前两个词的模型为三元模型。以此类推，我们可以建立四元模型，五元模型,... 一直到通用的 N 元模型。越往后，概率分布的计算复杂度越高。当然算法的原理是类似的。

在实际应用中，N 一般都较小，一般都小于 4，主要原因是 N 元模型概率分布的空间复杂度为 $O(|V|^N)$ ，其中 $|V|$ 为语料库大小，而 N 为模型的元数，当 N 增大时，复杂度呈指数级的增长。N 元模型的分词方法虽然很好，但是要在实际中应用也有很多问题，首先，某些生僻词，或者相邻分词联合分布在语料库中没有，概率为 0。这种情况我们一般会使用拉普拉斯平滑，即给它一个较小的概率值避免计算问题。第二个问题是如果句子长，分词有很多情况，计算量也非常大，这时我们就能偶使用维特比算法来优化算法¹时间复杂度。

2.1.3 小结

对于文本挖掘中需要的分词功能，一般我们会用现有的工具。简单的英文分词不需要任何工具，通过空格和标点符号就可以分词了，而进一步的英文分词推荐使用 nltk。对于中文分词，则推荐用结巴分词 (jieba)。分词是文本挖掘的预处理的重要的一步，分词完成后，我们可以继续做一些其他的特征工程，比如向量化 (vectorize)，随后就能够进

¹具体的算法过程会在后面的章节会详细说明

行下面的分析工作。

2.2 HMM 模型与维比特算法

2.2.1 HMM 模型的定义

对于 HMM 模型，首先我们假设 Q 是所有可能的隐藏状态的集合， V 是所有可能的观测状态的集合，即：

$$Q = q_1, q_2, \dots, q_N, V = v_1, v_2, \dots, v_M$$

其中， N 是可能的隐藏状态数， M 是所有的可能的观察状态数。对于一个长度为 T 的序列， I 对应的状态序列， O 是对应的观察序列，即：

$$I = i_1, i_2, \dots, i_T, O = o_1, o_2, \dots, o_T$$

其中，任意一个隐藏状态 $i_t \in Q$ ，任意一个观察状态 $o_t \in V$ HMM 模型做了两个很重要的假设如下：

1. 齐次马尔科夫链假设。即任意时刻的隐藏状态只依赖于它前一个隐藏状态。

当然这样假设有点极端，因为很多时候我们的某一个隐藏状态不仅仅只依赖于前一个隐藏状态，可能是前两个或者是前三个。但是这样假设的好处就是模型简单，便于求解。如果在时刻 t 的隐藏状态是 $i_t = q_i$ ，在时刻 $t+1$ 的隐藏状态是 $i_{t+1} = q_j$ ，则从时刻 t 到时刻 $t+1$ 的 HMM 状态转移概率 a_{ij} 可以表示为：

$$\begin{aligned} a_{ij} &= P(i_{t+1} = q_j | i_t = q_i, i_{t-1}, \dots, i_1, o_1, o_2, \dots, o_t) \\ &= P(i_{t+1} = q_j | i_t = q_i) \end{aligned} \quad (6)$$

这样 a_{ij} 可以组成马尔科夫链的状态转移矩阵 A ：

$$A = [a_{ij}]_{N \times N}$$

2. 观测独立性假设。即任意时刻的观察状态只仅仅依赖于当前时刻的

隐藏状态.

这也是一个为了简化模型的假设。如果在时刻 t 的隐藏状态是 $i_t = q_j$, 而对应的观察状态为 $o_t = v_k$, 则该时刻观察状态 v_k 在隐藏状态 q_j 下生成的概率为 $b_j(k)$, 满足:

$$\begin{aligned} b_j(k) &= P(o_t = v_k | i_t = q_j, i_{t-1}, \dots, i_1, o_1, o_2, \dots, o_t) \\ &= P(o_t = v_k | i_t = q_j) \end{aligned} \quad (7)$$

这样 $b_j(k)$ 可以组成观测状态生成的概率矩阵 B :

$$B = [b_j(k)]_{N \times M}$$

除此之外, 我们需要一组在时刻 $t = 1$ 的隐藏状态概率分布 Π :

$$\Pi = [\pi(i)]_N$$

其中 $\pi(i) = P(i_1 = q_i)$

一个 HMM 模型, 可以由隐藏状态初始概率分布 Π , 状态转移概率矩阵 A 和观测状态概率矩阵 B 决定。 Π, A 决定状态序列, B 决定观测序列。因此, HMM 模型可以由一个三元组 λ 表示如下:

$$\lambda = (A, B, \Pi)$$

HMM 模型一共有三个经典的问题需要解决:

1. 评估观察序列概率。即给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = o_1, o_2, \dots, o_T$, 计算在模型 λ 下观测序列 O 出现的概率 $P(O|\lambda)$ 。这个问题的求解需要用到前向后向算法。
2. 模型参数学习问题。即给定观测序列 $O = o_1, o_2, \dots, o_T$, 估计模型 $\lambda = (A, B, \Pi)$ 的参数, 使该模型下观测序列的条件概率 $P(O|\lambda)$ 最大。这个问题的求解需要用到基于 EM 算法的鲍姆-韦尔奇算法。
3. 预测问题, 也称为解码问题。即给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = o_1, o_2, \dots, o_T$, 求给定观测序列条件下, 最可能出现的对应的状

态序列，这个问题的求解需要用到基于动态规划的维特比算法。

在本次实验中需要使用的是维比特算法与向前向后算法，关于鲍姆-韦耳奇算法这里就不过多介绍。

2.2.2 问题的引入

问题描述：已知 HMM 模型的参数 $\lambda = (A, B, \Pi)$ 。其中 A 是隐藏状态转移概率的矩阵，B 是观测状态生成概率的矩阵， Π 是隐藏状态的初始概率分布。同时我们也已经得到了观测序列 $O = o_1, o_2, \dots, o_T$ ，现在要求观测序列 O 在模型 λ 下出现的条件概率 $P(O|\lambda)$ 。

如果直接通过暴力求解：通过列举出所有可能出现的长度为 T 的隐藏序列 $I = i_1, i_2, \dots, i_T$ ，分布求出这些隐藏序列与观测序列 $O = o_1, o_2, \dots, o_T$ 的联合概率分布 $P(O, I|\lambda)$ ，这样可以很容易的求出边缘分布 $P(O|\lambda)$ 了。

具体暴力求解的过程如下：首先，任意一个隐藏序列 $I = i_1, i_2, \dots, i_T$ 出现的概率是：

$$P(I|\lambda) = \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \dots a_{i_{T-1} i_T}$$

对于固定的状态序列 $I = i_1, i_2, \dots, i_T$ ，要求的观察序列 $O = o_1, o_2, \dots, o_T$ 出现的概率是：

$$P(O|I, \lambda) = b_{i_1}(o_1) b_{i_2}(o_2) \dots b_{i_T}(o_T)$$

则 O 和 I 联合出现的概率是：

$$P(O, I|\lambda) = P(I|\lambda) P(O|I, \lambda) = \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) \dots a_{i_{T-1} i_T} b_{i_T}(o_T)$$

然后求边缘概率分布，即可得到观测序列 O 在模型 λ 下出现的条件概率 $P(O|\lambda)$ ：

$$P(O|\lambda) = \sum_I P(O, I|\lambda) = \sum_{i_1, i_2, \dots, i_T} \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) \dots a_{i_{T-1} i_T} b_{i_T}(o_T)$$

虽然上述方法有效，但是如果隐藏状态数 N 非常多的那就麻烦了，此

时预测状态有 N^T 种组合，算法的时间复杂度是 $O(TN^T)$ 阶的。因此对于一些隐藏状态数极少的模型，可以用暴力求解法来得到观测序列出现的概率，但是如果隐藏状态多，则上述算法太耗时，导致需要寻找其他简洁的算法。前向后向算法就是使用较低的时间复杂度情况下求解这个问题的。

2.2.3 前向算法

前向算法本质上属于动态规划的算法，首先需要通过找到局部状态递推的公式，这样一步步的从子问题的最优解拓展到整个问题的最优解在前向算法中，通过定义“前向概率”来定义动态规划的这个局部状态。

定义：时刻 t 时隐藏状态为 q_i ，观测状态的序列为 o_1, o_2, \dots, o_t 的概率为前向概率。记为：

$$\alpha_t(i) = P(o_1, o_2, \dots, o_t, i_t = q_i | \lambda)$$

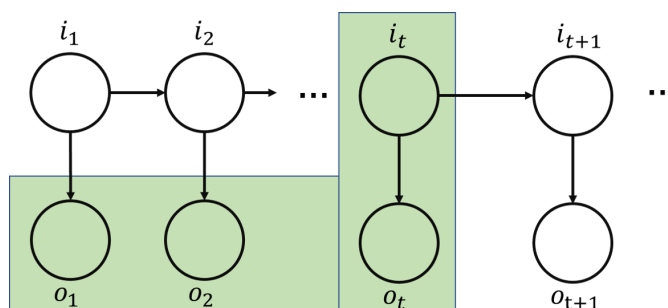


图 3: 前向概率图示

现在假设已经找到了在时刻 t 时各个隐藏状态的前向概率，现在需要递推出时刻 $t+1$ 时各个隐藏状态的前向概率：

$$\begin{aligned} \alpha_{t+1}(j) &= P(o_1, o_2, \dots, o_t, o_{t+1}, i_{t+1} = q_j | \lambda) \\ &= \sum_{i=1}^N P(o_1, o_2, \dots, o_t, o_{t+1}, i_{t+1} = q_j, i_t = q_i | \lambda) \end{aligned}$$

$$\begin{aligned}
&= \sum_{i=1}^N P(o_{t+1}|o_1, \dots, o_t, i_{t+1} = q_j, i_t = q_i, \lambda) P(o_1, \dots, o_t, i_{t+1} = q_j, i_t = q_i | \lambda) \\
&= \sum_{i=1}^N P(o_{t+1}|i_{t+1} = q_j, \lambda) * P(o_1, \dots, o_t, i_{t+1} = q_j, i_t = q_i | \lambda) \\
&= \sum_{i=1}^N P(o_{t+1}|i_{t+1} = q_j, \lambda) P(i_{t+1} = q_j | o_1, \dots, o_t, i_t = q_i, \lambda) P(o_1, \dots, o_t, i_t = q_i | \lambda) \\
&= \sum_{i=1}^N P(o_{t+1}|i_{t+1} = q_j, \lambda) P(i_{t+1} = q_j | i_t = q_i, \lambda) \alpha_t(i) \\
&= \sum_{i=1}^N b_j(o_{t+1}) a_{ij} \alpha_t(i)
\end{aligned}$$

至此就求解得到了关于 α 的递推公式：

$$\alpha_{t+1}(j) = b_j(o_{t+1}) \sum_{i=1}^N a_{ij} \alpha_t(i) \quad (8)$$

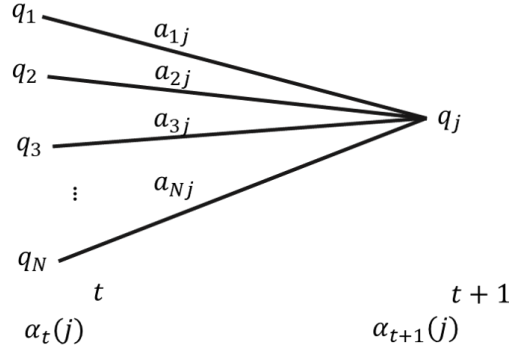


图 4: 前向概率递推公式图示

动态规划从时刻 1 开始，到时刻 T 结束，由于 $\alpha_T(i)$ 表示在时刻 T 观测序列为 o_1, o_2, \dots, o_T ，并且时刻 T 隐藏状态 q_i 的概率，只要将所有隐藏状态对应的概率相加，即 $\sum_{i=1}^N \alpha_T(i)$ 就得到了在时刻 T 观测序列为 o_1, o_2, \dots, o_T 的概率。

于是前向算法总结如下：

1. 输入：HMM 模型 $\lambda = (A, B, \Pi)$ ，观测序列 $O = (o_1, o_2, \dots, o_T)$

输出：输出观测序列概率 $P(O|\lambda)$

2. 计算时刻 1 的各个隐藏状态前向概率：

$$\alpha_1(i) = \pi_i b_i(o_1), i = 1, 2, \dots, N$$

3. 递推时刻 2, 3, ..., T 时刻的前向概：

$$\alpha_{t+1}(i) = \left[\sum_{j=1}^N \alpha_t(j) a_{ij} \right] b_i(o_{t+1})$$

4. 最终计算结果：

$$P(O|\lambda) = \sum_{i=1}^N \alpha_T(i)$$

从递推公式可以看出，前向算法时间复杂度是 $O(TN^2)$ ，比暴力解法的时间复杂度 $O(TN^T)$ 少了几个数量级。

2.2.4 向后算法

后向算法和前向算法非常类似，都是用的动态规划，唯一的区别是选择的局部状态不同，后向算法用的是“后向概率”：

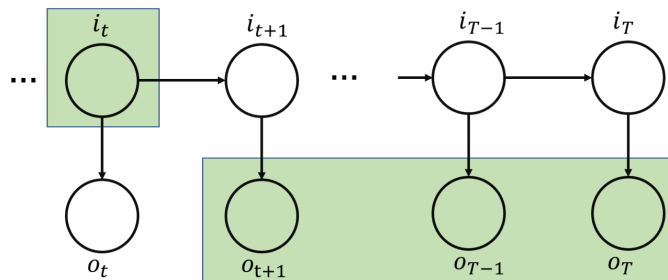


图 5: 后向概率图示

定义: 时刻 t 隐藏状态为 q_i , 从时刻 $t+1$ 到最后时刻 T 的观测状态的序列为 $o_{t+1}, o_{t+2}, \dots, o_T$ 的概率为后向概率。记为:

$$\beta = P(o_{t+1}, o_{t+2}, \dots, o_T | i_t = q_i, \lambda)$$

同前向算法一样, 首先来推导出关于参数 β 的递推式:

$$\begin{aligned} \beta_t(i) &= P(o_{t+1}, \dots, o_T | i_t = q_i) \\ &= \sum_{j=1}^N P(o_{t+1}, \dots, o_T, i_{t+1} = q_j | i_t = q_i) \\ &= \sum_{j=1}^N P(o_{t+1}, \dots, o_T | i_{t+1} = q_j, i_t = q_i) P(i_{t+1} = q_j | i_t = q_i) \\ &= \sum_{j=1}^N P(o_{t+1}, \dots, o_T | i_{t+1} = q_j) a_{ij} \\ &= \sum_{j=1}^N P(o_{t+1} | o_{t+2}, \dots, o_T, i_{t+1} = q_j) P(o_{t+2}, \dots, o_T | i_{t+1} = q_j) a_{ij} \\ &= \sum_{j=1}^N b_j(o_{t+1}) \beta_{t+1}(j) a_{ij} \end{aligned}$$

这样求得后向概率的递推关系式如下:

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \quad (9)$$

在推导出来了参数 β 的递推公式之后, 再推导观测序列概率 $P(O | \lambda)$

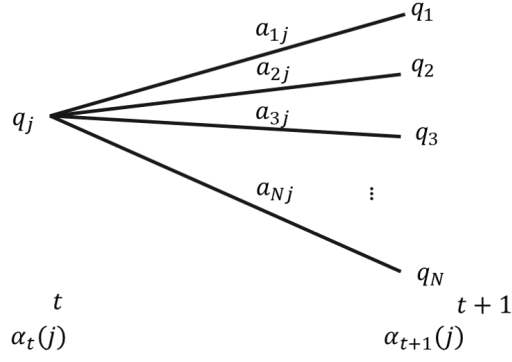


图 6: 后向概率递推公式图示

与后向概率 $\beta_t(i)$ 的关系:

$$\begin{aligned}
 P(O|\lambda) &= P(o_1, o_2, \dots, o_T|\lambda) \\
 &= \sum_{i=1}^N P(o_1, o_2, \dots, o_T, i_1 = q_i|\lambda) \\
 &= \sum_{i=1}^N P(o_1, o_2, \dots, o_T|i_1 = q_i, \lambda)P(i_1 = q_i|\lambda) \\
 &= \sum_{i=1}^N P(o_1|o_2, \dots, o_T, i_1 = q_i, \lambda)P(o_1, o_2, \dots, o_T|i_1 = q_i, \lambda)\pi_i \quad (10) \\
 &= \sum_{i=1}^N P(o_1|i_1 = q_i, \lambda)\beta_1(i)\pi_i \\
 &= \sum_{i=1}^N b_i(o_1)\beta_1(i)\pi_i
 \end{aligned}$$

总结后向算法的流程, 注意下和前向算法的相同点和不同点:

-
1. **输入:** HMM 模型 $\lambda = (A, B, \Pi)$, 观测序列 $O = (o_1, o_2, \dots, o_T)$
输出: 输出观测序列概率 $P(O|\lambda)$
 2. 计算时刻 T 的各个隐藏状态的后向概率:

$$\beta_T(i) = 1, i = 1, 2, \dots, N$$

3. 递推时刻 $T-1, T-2, \dots$ 时刻的前向概:

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+2}(j), i = 1, 2, \dots, N$$

4. 最终计算结果:

$$P(O|\lambda) = \sum_{i=1}^N \pi_i b_i(o_1) \beta_1(i)$$

此时后向算法时间复杂度仍然是 $O(TN^2)$ 。

2.2.5 维比特算法

HMM 模型的解码问题最常用的算法是维特比算法，同时维特比算法是一个通用的求序列最短路径的动态规划算法，本节用维特比算法来解码 HMM 的最可能隐藏状态序列。

在 HMM 模型的解码问题中，给定模型 $\lambda = (A, B, \Pi)$ 和观测序列 $O = o_1, o_2, \dots, o_T$ ，求给定观测序列 O 条件下，最可能出现的对应的状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$ ，即 $P(I^*|O)$ 要最大化。

一个可能的近似解法是求出观测序列 O 在每个时刻 t 最可能的隐藏状态 i_t^* ，然后得到一个近似的隐藏状态序列 $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$ 。在给定模型 λ 和观测序列 O 时，在时刻 t 处于状态 q_i 的概率是 $\gamma_t(i)$:

$$\gamma_t(i) = \frac{\alpha_t(i) \beta_t(i)}{\sum_{j=1}^N \alpha_t(j) \beta_t(j)}$$

在每一时刻 t 最有可能的状态 i_t^* 是:

$$i_t^* = \arg \max_{1 \leq i \leq N} [\gamma_t(i)], t = 1, 2, \dots, T$$

从而就能够求得: $I^* = \{i_1^*, i_2^*, \dots, i_T^*\}$ 。近似算法很简单，但是却不能保证预测的状态序列是整体是最可能的状态序列，因为预测的状态序列中某些相邻的隐藏状态可能存在转移概率为 0 的情况。而维特比算法可以将 HMM 的状态序列作为一个整体来考虑，避免近似算法的问题，下

面就介绍维特比算法进行 HMM 解码的方法。

维特比算法是一个通用的解码算法，是基于动态规划的求序列最短路径的方法。既然是动态规划算法，那么就需要找到合适的局部状态，以及局部状态的递推公式。在 HMM 中，维特比算法定义了两个局部状态用于递推。

第一个局部状态是在时刻 t 隐藏状态为 i 所有可能的状态转移路径 i_1, i_2, \dots, i_t 中的概率最大值。记为 $\delta_t(i)$:

$$\delta_t(i) = \arg \max_{i_1, i_2, \dots, i_{t-1}} P(i_t = q_i, i_1, i_2, \dots, i_{t-1}, o_1, o_2, \dots, o_t | \lambda), i = 1, 2, 3 \dots N$$

由 $\delta_t(i)$ 的定义可以得到 δ 的递推表达式，由于在递推式多是在模型 λ 下计算的，在下式中省略 λ :

$$\begin{aligned} \delta_{t+1}(i) &= \arg \max_{i_1, i_2, \dots, i_t} P(i_{t+1} = q_j, i_1, i_2, \dots, i_t, o_1, o_2, \dots, o_{t+1}) \\ &= \arg \max_{i_1, i_2, \dots, i_t} P(o_{t+1} | i_{t+1} = q_j, i_1, \dots, i_t, o_1, \dots, o_t) P(i_{t+1} = q_j, i_1, \dots, i_t, o_1, \dots, o_t) \\ &= \arg \max_{i_1, i_2, \dots, i_t} P(o_{t+1} | i_{t+1} = q_j) P(i_{t+1} = q_j | i_1, \dots, i_t, o_1, \dots, o_t) P(i_1, \dots, i_t, o_1, \dots, o_t) \\ &= \arg \max_{i_1, i_2, \dots, i_t} b_i(o_{t+1}) a_{ji} \delta_t(i) \end{aligned}$$

第二个局部状态由第一个局部状态递推得到。

定义: 在时刻 t 隐藏状态为 i 的所有单个状态转移路径 $(i_1, i_2, \dots, i_{t-1}, i_t)$ 中, 概率最大的第 $t-1$ 个节点的隐藏状态为 $\Psi_t(i)$, 其递推表达式可以表示为:

$$\Psi_t(i) = \arg \max_{i_1, i_2, \dots, i_t} [\delta_{t-1}(j) a_{ji}] \quad (11)$$

在上述定义之后，总结维特比算法如下:

1. 输入: HMM 模型 $\lambda = (A, B, \Pi)$, 观测序列 $O = (o_1, o_2, \dots, o_T)$

输出: 最有可能的隐藏状态序列 $I^* = i_1^*, i_2^*, \dots, i_T^*$

2. 初始化局部状态：

$$\delta_1(i) = \delta_1(i) = \pi_i b_i(o_1), i = 1, 2, \dots, N$$

$$\Psi_t(i) = \Psi_1(i) = 0, i = 1, 2, \dots, N$$

3. 进行动态规划递推时刻 $t = 2, 3, \dots, T$ 时刻的局部状态：

$$\delta_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j) a_{ji}] b_i(o_t)$$

$$\Psi_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j) a_{ji}]$$

4. 计算时刻 T 最大的 $\delta_t(i)$, 即为最可能隐藏状态序列出现的概率。计算时刻 T 最大的 $\Psi_t(i)$, 即为时刻 T 最可能的隐藏状态。

$$P^* = \max_{1 \leq j \leq N} \delta_T(i)$$

$$i_t^* = \arg \max_{1 \leq j \leq N} [\delta_T(i)]$$

5. 利用局部状态 $\Psi_T(i)$ 开始回溯。对于 $t = T-1, T-2, \dots, 1$ ：

$$i_t^* = \Psi_{t+1}(i_{t+1}^*)$$

最终得到最有可能的隐藏状态序列 $I^* = i_1^*, i_2^*, \dots, i_T^*$

2.2.6 维比特算法举例

维特比算法采用的是动态规划来解决这个最优分词问题的，动态规划要求局部路径也是最优路径的一部分，很显然这个问题是成立的。首先看一个简单的分词例子：“人生如梦境”。它的可能分词可以用下面的概率图表示：

图中的箭头为通过统计语料库而得到的对应的各分词位置 BEMS(开始位置, 结束位置, 中间位置, 单词) 的条件概率。比如 $P(\text{生} | \text{人})=0.17$ 。有了这个图，维特比算法需要找到从 Start 到 End 之间的一条最短路

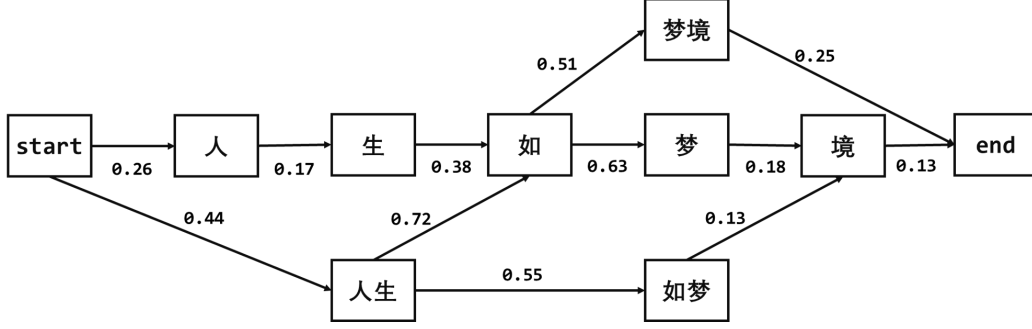


图 7: 分词概率图

径。对于在 End 之前的任意一个当前局部节点，我们需要得到到达该节点的最大概率 δ ，和记录到达当前节点满足最大概率的前一节点位置 Ψ 。初始化基础有：

$$\begin{aligned}\delta(\text{人}) &= 0.26, \Psi(\text{人}) = \text{Start}, \\ \delta(\text{人生}) &= 0.44, \Psi(\text{人生}) = \text{Start}\end{aligned}$$

对于节点”生”，它只有一个前向节点，因此有：

$$\delta(\text{生}) = \delta(\text{人})P(\text{生} | \text{人}) = 0.0442, \Psi(\text{生}) = \text{人}$$

对于节点”如”，就稍微复杂一点了，因为它有多个前向节点，就需要计算出”如”概率最大的路径：

$$\begin{aligned}\delta(\text{如}) &= \max\{\delta(\text{生})P(\text{如} | \text{生}), \delta(\text{人生})P(\text{如} | \text{人生})\} \\ &= \max\{0.016796, 0.3168\} = 0.3168, \\ \Psi(\text{如}) &= \text{人生}\end{aligned}$$

类似的方法可以用于其他节点如下：

$$\begin{aligned}\delta(\text{如梦}) &= \delta(\text{人生})P(\text{如梦} | \text{人生}) = 0.242, \Psi(\text{如梦}) = \text{人生} \\ \delta(\text{梦}) &= \delta(\text{如})P(\text{梦} | \text{如}) = 0.199584, \Psi(\text{梦}) = \text{如} \\ \delta(\text{梦境}) &= \delta(\text{如})P(\text{梦境} | \text{如}) = 0.161568, \Psi(\text{梦境}) = \text{如} \\ \delta(\text{境}) &= \max\{\delta(\text{梦})P(\text{境} | \text{梦}), \delta(\text{如梦})P(\text{境} | \text{如梦})\}\end{aligned}$$

$$\begin{aligned}
&= \max\{0.03592512, 0.03146\} = 0.03592512, \Psi(\text{境}) = \text{梦} \\
\delta(\text{end}) &= \max\{\delta(\text{境})P(\text{境}|\text{end}), \delta(\text{梦境})P(\text{梦境}|\text{end})\} \\
&= \max\{0.040392, 0.0046702656\} = 0.040392, \Psi(\text{end}) = \text{梦境}
\end{aligned}$$

然后通过 $\Psi(\text{end}) = \text{梦境}$ 回溯路径就能够得到：

$$\Psi(\text{end}) = \text{梦境} \rightarrow \Psi(\text{梦境}) = \text{如} \rightarrow \Psi(\text{如}) = \text{人生}$$

从而最终的分词结果为”人生/如/梦境”。

2.2.7 总结

如果仔细比较例子中的维特比算法，与讲解的算法原理相比存在一些差异。主要原因是在中文分词时，没有观察状态和隐藏状态的区别，只有一种状态。但是维特比算法的核心是定义动态规划的局部状态与局部递推公式，这一点在中文分词维特比算法和 HMM 的维特比算法是相同的，也是维特比算法的精华所在。

维特比算法也是寻找序列最短路径的一个通用方法，和 dijkstra 算法有些类似，但是 dijkstra 算法并没有使用动态规划，而是贪心算法。同时维特比算法仅仅局限于求序列最短路径，而 dijkstra 算法是通用的求最短路径的方法。

2.3 SAG 模型

2.3.1 SAG 定义

图由顶点和连接这些顶点的边所构成。每条边都带有从一个顶点指向另一个顶点的方向的图为有向图。有向图中的道路为一系列的边，系列中每条边的终点都是下一条边的起点。如果一条路径的起点是这条路径的终点，那么这条路径就是一个环。有向无环图即为没有环出现的有向图。在以蓝线标识的有向无环图中，添加红线从而得到其传递闭包当存在一条从顶点 u 到顶点 v 的路径时，顶点 v 被称作是从顶点 u 可达的。每个顶点都是从自身可达的（通过一条没有边的路径）。如果一个

顶点可以从一个非平凡路径（一条由一个或更多边组成的路径）到达自身，那么这条路径就是一个环。因此，有向无环图也可以被定义为没有顶点可以通过非平凡路径到达自身的图。

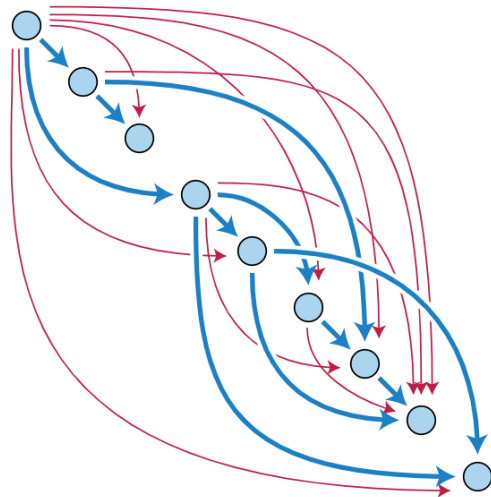


图 8: 在以蓝线标识的有向无环图中，添加红线从而得到其传递闭包

2.3.2 SAG 分词原理

使用 SAG 分词划分为三步：

1. 首先是基于统计词典构造前缀词典，如统计词典中的词“北京大学”的前缀分别是“北”、“北京”、“北京大”；词“大学”的前缀是“大”。统计词典中所有的词形成的前缀词典如下所示：

1	北京大学 2053
2	北京大 0
3	大学 20025
4	去 123402
5	玩 4207
6	北京 34488
7	北 17860
8	京 6583
9	大 144099
10	学 17482

2. 然后基于前缀词典，对输入文本进行切分，对于“去”，没有前缀，那么就只有一种划分方式；对于“北”，则有“北”、“北京”、“北

京大学”三种划分方式；对于“京”，也只有一种划分方式；对于“大”，则有“大”、“大学”两种划分方式，依次类推，可以得到每个字开始的前缀词的划分方式。

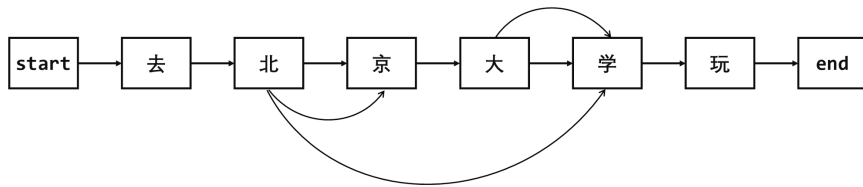


图 9: 基于前缀词典的文本划分

3. 在得到所有可能的切分方式构成的有向无环图后，从起点到终点存在多条路径，多条路径也就意味着存在多种分词结果，因此需要计算最大概率路径，也即按照这种方式切分后的分词结果的概率最大。在计算最大概率路径时，有向无环图的方向是从前向后指向，对于一个节点，只知道这个节点会指向后面哪些节点，很难直接知道有哪些前面的节点会指向这个节点。在采用动态规划计算最大概率路径时，每到达一个节点，它前面的节点到终点的最大路径概率已经计算出来。

构建出的有向无环图 DAG 的每个节点，都是带权的，对于在前缀词典里面的词语，其权重就是它的词频；我们想要求得 $route = (w_1, w_2, w_3, \dots, w_n)$ ，使得 $\sum weight(w_i)$ 最大。

- 任意通过 W_i 到达 W_j 的路径的权重 = 该路径通过 W_i 的路径权重 + W_j 的权重，也即 $R_{i \rightarrow j} = R_i + weight(j)$
- 任意通过 W_i 到达 W_k 的路径的权重 = 该路径通过 W_i 的路径权重 + W_k 的权重，也即 $R_{i \rightarrow k} = R_i + weight(k)$

即对于拥有公共前驱节点 W_i 的节点 W_j 和 W_k ，需要重复计算达到 W_i 的路径的概率。对于整个句子的最优路径 R_{max} 和一个末端节点 W_x ，对于其可能存在的多个前驱 $W_i, W_j, W_k \dots$ ，设到达 W_i, W_j, W_k 的最大路径分别是 $R_{maxi}, R_{maxj}, R_{maxk}$ ，有：

$$R_{max} = \max\{R_{maxi}, R_{maxj}, R_{maxk}, \dots\} + weight(W_x)$$

于是, 问题转化为, 求解 $R_{maxi} R_{maxj} R_{maxk} \dots$ 等, 由此组成了最优子结构, 子结构里面的最优解是全局的最优解的一部分。原问题的状态转移方程式就是上式。

2.3.3 小结

在本节中我们介绍了有向无环图分词的基本原理, 将其化分为三步, 首先根据统计词典构造前缀词典, 然后根据前缀词典划分文本输入, 最后根据文本输入寻找概率最大的路径。在分词过程上与 HMM 有一定的相识之处, 都是构造图模型, 然后使用动态规划算法进行求解。因 HMM 模型对模型进行了较强的条件假设, 在单独分词性能不佳。但是在现有的 jieba 分词工具中任然保留了 HMM 分词的部分, 他将 HMM 与 DAG 两个模型的结果融合, 融合了 DAG 稳定的构词能力和 HMM 的新词发现能力, 在分词上效果就非常不错了。之所以在研究 HMM 分词后还研究 DAG 模型也是因为读取了 jieba 的源码提出的想法。

2.4 朴素贝叶斯模型

朴素贝叶斯 (naive Bayes) 法是基于贝叶斯定理与特征条件独立假设的分类方法”。对于给定的训练数据集, 首先基于特征条件独立假设学习输入/输出的联合概率分布; 然后基于此模型, 对给定的输入 x , 利用贝叶斯定理求出后验概率最大的输出 y 。朴素贝叶斯法实现简单, 学习与预测的效率都很高, 是一种常用的方法。接下来就介绍该算法。

2.4.1 朴素贝叶斯方法

设输入空间 $\chi \in R$ 为 n 维向量的集合, 输出空间为类标记集合 $\Upsilon = c_1, c_2, \dots, c_n$ 输入为特征向量 $x \in \chi$, 输出为类标记 (class label) $y \in \Upsilon$. X 是定义在输入空间 χ 上的随机向量, Y 是定义在输出空间 D 上的随机变量. $P(X, Y)$ 是 X 和 Y 的联合概率分布. 训练数据集

$$T = \{(x_1, y_1), (x_2, y_2), \dots (x_N, y_N)\}$$

在给定的数据集下我们能够求出 X 和 Y 的联合概率分布 $P(X, Y)$, 具体来说学习先验概率:

$$P(Y = c_k) (k = 1, 2, \dots, n) \quad (12)$$

条件概率:

$$P(X = x|Y = c_k) = P(X^{(1)} = x^{(1)}, \dots, X^{(N)} = x^{(N)}|Y = c_k) \quad (13)$$

于是就能得到联合概率函数:

$$P(X, Y) = \sum_{i=0}^N P(X = x|Y = c_k)P(Y = c_k) \quad (14)$$

在计算条件概率中 $P(X = x|Y = c_k)$ 是一个非常大的指数级参数, 在求取过程中非常复杂。于是, 求解过程中给定一个条件独立的假设: 分类的特征在分类确定的条件下, 条件都是独立的, 即

$$\begin{aligned} P(X = x|Y = c_k) &= P(X^{(1)} = x^{(1)}, \dots, X^{(N)} = x^{(N)}|Y = c_k) \\ &= \prod_{j=1}^N P(X^j = x^j|Y = c_k) \end{aligned} \quad (15)$$

2.4.2 模型推导

在已知 $X = (x^{(1)}, x^{(2)}, \dots, x^{(n)})$ 的情况下, 需要求得 $P(Y = C_k)$ 的概率, 即: $P(X|Y)$. 根据朴素贝叶斯法, $P(X|Y) = \frac{P(X, Y)}{P(Y)}$, 将(12)、(14)、(15)带入得到:

$$\begin{aligned} P(Y = C_k|X = x) &= \frac{P(X = x|Y = c_k)P(Y = c_k)}{P(Y = c_k)} \\ &= \frac{P(X = x|Y = c_k)P(Y = c_k)}{\sum_k P(X = x|Y = c_k)P(Y = c_k)} \\ &= \frac{P(Y = c_k) \prod_{j=1}^n P(X^j = x^j|Y = c_k)}{\sum_k P(Y = c_k) \prod_{j=1}^n P(X^j = x^j|Y = c_k)} \end{aligned} \quad (16)$$

自此(16)所有的值都能通过已知数据获得。于是朴素贝斯分类器(17)可表示为：

$$y = f(x) = \arg \max_{c_k} \frac{P(Y = c_k) \prod_{j=0}^n P(X^j = x^j | Y = c_k)}{\sum_k P(Y = c_k) \prod_{j=1}^n P(X^j = x^j | Y = c_k)} \quad (17)$$

对于所有的分类而言，分类器的分母都是相同的，所以分类器可简化(18)为：

$$y = f(x) = \arg \max_{c_k} P(Y = c_k) \prod_{j=1}^n P(X^j = x^j | Y = c_k) \quad (18)$$

2.4.3 最大化后验概率的含义

朴素贝叶斯方法属于分类问题，我们选择 0-1 损失函数衡量它的损失：

$$L(Y, f(x)) = \begin{cases} 1, Y \neq f(x) \\ 0, Y = f(x) \end{cases}$$

其中 $f(x)$ 表示分类决策函数，此时经验风险函数可表示为：

$$R_{exp}(f) = E[L(Y, f(x))] = E_X \sum_{i=1}^K [L(c_k, f(x))] P(c_k | X)$$

最小化经验损失函数就是对每一个 x 进行最小化，

$$\begin{aligned} \min R_{exp}(f) &= \arg \min_{c_k} \sum_{i=1}^k L(c_k, y) P(c_k | X) \\ &= \arg \min_{c_k} \sum_{i=1}^k P(y \neq c_k | X) \\ &= \arg \min_{c_k} \sum_{i=1}^k 1 - P(y = c_k | X) \end{aligned}$$

$$= \arg \max_{c_k} \sum_{i=1}^k P(y = c_k | X)$$

也就是说，最小化经验风险函数，与最大化决策分类函数是一样的，这也就是朴素贝叶斯方法的原理。

2.4.4 多项式朴素贝叶斯 (Multinomial Naive Bayes)

在上述的决策函数中， $P(X = x | Y = c_k)$ 是不知道的，于是不同的求取这个条件概率就衍生了不同的算法，如高斯朴素贝叶斯 (GaussianNB)、伯努利朴素贝叶斯 (Bernoulli Naive Bayes)、多项式朴素贝叶斯方法 (Multinomial Naive Bayes)，在此只介绍多项式朴素贝叶斯方法：该方法是应用于文本分类 (Text Classification) 的 2 个经典朴素贝叶斯变体之一。它的特征数据服从多项式分布，使用单词计数向量 (Word Count Vector) 来表示输入数据；分布参数由参数向量表示：

$$\theta_y = (\theta_{y1}, \theta_{y2}, \dots, \theta_{yn})$$

其中， n 表示特征个数，例如，在文本分类中， n 表示单词容量。 θ_{yi} 表示特征 i 出现在属于类别 y 的样本 x 中的概率 $P(x_i | y)$ 。由于特征条件分布的独立性假设，参数 θ_y 的每个分量都可以单独使用最大似然方法进行估算，常用情况下，使用平滑版本：

$$P(x_i | y) = \theta_{yi}^{\wedge} = \frac{N_{yi} + \alpha}{N_y + n\alpha}$$

其中， $N_{yi} = \sum_{x \in T} x_i$ 是训练集 T 中，特征 i 出现在类别 y 中的个数， $N_y = \sum_{i=1}^n N_{yi}$ 是类别 y 中所有特征的个数总和， n 表示特征 x_i 的取值个数。上式中， $\alpha \geq 0$ ，当 $\alpha = 0$ 时，它等价于最大似然估计。 $\alpha > 0$ 可以防止待估计的概率值出现 0 的情况，避免对后续计算产生影响。当 $\alpha = 1$ 时，该平滑公式被称为拉普拉斯平滑 (Laplace Smoothing)，当 $\alpha < 1$ 时，被称为 Lidstone 平滑。

2.4.5 小结

在这一节简单讲述了朴素贝叶斯的基本原理，讲述了最大后验概率的含义，在最后由于条件概率的生产不同，也产生了不同的朴素贝叶斯方法，在本文中重点强调的多项式朴素贝叶斯方法。从下章开始，我们将针对具体的场景，具体的数据进行具体的代码编写与相关步骤的讲解。

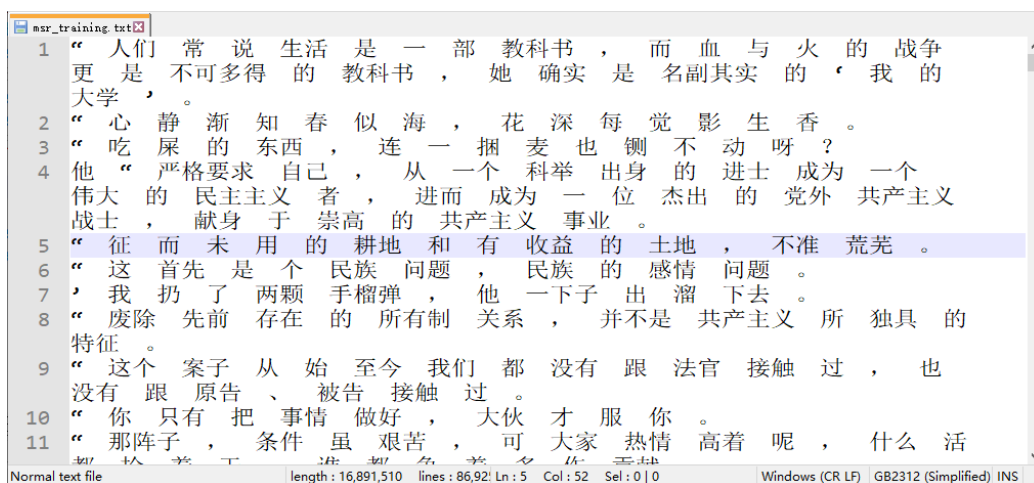
3 代码实现

在 2.1 节讲述了中文分词的原理与三种中文分词的方法分别使用的 HMM 模型，DAG 模型与 Jie ba 分词库，在下面的过程中，我们将实现这三种分词，并对比它的效果。

3.1 HMM 中文分词

3.1.1 数据说明

在实验中使用的数据为第二届国际汉语分词四家单位提供的测试语料 (Academia Sinica、City University、Peking University、Microsoft-Research) 中使用了 MicrosoftResearch 的数据集 (./data/msr_training.txt), 部分数据如下：



```
msr_training.txt
1 “人们常说生活是一部教科书，而血与火的战争
  更是不可多得的教科书，她确实是名副其实的‘我的
  大学’。
2 “心静渐知春似海，花深每觉影生香。
3 “吃屎的东西，连一捆麦也铡不动呀？
4 他“严格要求自己，从一个科举出身的进士成为一个
  伟大的民主主义者，进而成为一位杰出的党外共产主义
  战士，献身于崇高的共产主义事业。
5 “征而未用的耕地和有收益的土地，不准荒芜。
6 “这首先是个民族问题，民族的感情问题。
7 ‘我扔了两颗手榴弹，他一下子出溜下去。
8 “废除先前存在的所有制关系，并不是共产主义所独具的
  特征。
9 “这个案子从始至终我们都没有跟法官接触过，也
  没有跟原告、被告接触过。
10 “你只有把事情做好，大伙才服你。
11 “那阵子，条件虽艰苦，可大家热情高着呢，什么活
    和/ 松 苦 工 住 和 各 苦 多 住 苦 苦
```

图 10: 部分数据集展示

如图所示在文件的每一行中均有一句话，每句话中的单词均由空格分离开来，我们需要通过这些数据将得到发射矩阵、转移矩阵、初始矩阵，具体代码如下：

```
1 seg_stop_words = {
2     " ", " ", " ", "。", "“", "”", "? ", "! ", ": ", "《", "》", "\",
3     "; ", ". ", ' ', "' ", "—", "——", ", ", ". ", "?", "!", "`",
4     "~", "@", "#", "$", "%", "^", "&", "*", "(", ")", "-", "_",
5     "+", "=", "[", "]", "{", "}", "'", '"', "<", ">", "\\ ", "|"
6     "\r", "\n", "\t", " (", ") "}
7
8 def get_tags(src):
9     tags = []
10    if len(src) == 1:
11        tags = ['S']
12    elif len(src) == 2:
13        tags = ['B', 'E']
14    else:
15        m_num = len(src) - 2
16        tags.append('B')
17        for i in range(m_num):
18            tags.append('M')
19        tags.append('S')
20    return tags
21
22 filename = './data/msr_training.txt'
23
24 trans_mat = {"M":{"M":0., 'B':0, "S":0, "E":0}, "B":{"M":0, 'B':0, "S":0, "E":0}, "S":{"M":0., 'B':0, "S":0, "E":0}, "E":{"M":0., 'B':0, "S":0, "E":0}}
25 emit_mat = {"M":{ }, "B":{ }, "S":{ }, "E":{ }} # 发射矩阵
26 init_vec = {"M":0., 'B':0, "S":0, "E":0} # 初始矩阵
27 state_count = {"M":0., 'B':0, "S":0, "E":0}
28 with open(filename, 'r') as f:
29     lines = f.readlines()
30     for i in range(len(lines)):
31         words = lines[i].split(" ")
32         for index in range(len(words)):
33             if words[index] in seg_stop_words:
34                 end = None
35                 continue
36             tags = get_tags(words[index])
37             for tag_index in range(len(tags)):
38                 if(tag_index == len(tags) - 1):
39                     end = tags[tag_index]
40             # 初始矩阵
```

```

41         if tag_index == 0:
42             init_vec[tags[tag_index]] += 1
43         # 发射矩阵
44         # print(words[index][tag_index], emit_mat[tags[tag_index]].
45             keys())
46         if words[index] == "":
47             continue
48         if words[index][tag_index] in emit_mat[tags[tag_index]].keys
49             ():
50             state_count[tags[tag_index]] += 1
51         else:
52             state_count[tags[tag_index]] = 1
53         # 发射矩阵
54         if words[index][tag_index] in emit_mat[tags[tag_index]].keys
55             ():
56             emit_mat[tags[tag_index]][words[index][tag_index]] +=
57                 1
58         else:
59             emit_mat[tags[tag_index]][words[index][tag_index]] = 1
60         # 转移矩阵
61         if tag_index == 0:
62             if end != None:
63                 trans_mat[end][tags[tag_index]] += 1
64             else:
65                 trans_mat[tags[tag_index-1]][tags[tag_index]] += 1
66 data = {"trans_mat":trans_mat, "init_vec":init_vec, "emit_mat":emit_mat,"state_count":
67     state_count}
68 with open("./data\\dd.json", "w", encoding="utf-8") as fw:
69     json.dump(data, fw, ensure_ascii=False)
70 print("succeed")

```

3.1.2 HMM 模型 ADT 说明

上述的操作为我们的 HMM 模型打下了基础，在分词部分提供了发射矩阵、转移矩阵、初始矩阵，在使用维比特算法进行解码时中文分词与中文的词性标注原理是一样的，但是也存在部分差异，于是能够将重复的部分作为基类，分词模型与标注模型作为子类，在此给出 ADT：

- class HMMModel() HMM 基类
 - trans_mat 转移矩阵
 - emit_mat 发射矩阵

- `init_vec` 初始矩阵
 - `states` 状态信息，也就是观测信息
 - `inited` 是否初始化
 - `setup()` 初始化 `trans_mat`、`emit_mat`、`init_vec` 的大小
 - `save(filename='hmm.json', code='json')` 保存数据：`trans_mat`、`emit_mat`、`init_vec`、`state_count` 到 `filename` 中，不用每次预测时都训练模型。`filename` 为报文文件的文件名，`code` 为保存文件的格式，无返回值。
 - `load(filename='hmm.json', code='json')` 加载数据：`trans_mat`、`emit_mat`、`init_vec`、`state_count`。`filename` 加载文件的文件名，`code` 为加载文件的格式
 - `do_train(observe, states)` 通过观测序列与状态序列计算发射矩阵、转移矩阵、初始矩阵，`observe` 为观测序列，`states` 为状态序列
 - `get_prob()` 将统计数据转换成概率
 - `do_predict(c)` 维比特算法，对输入的 `sequence` 进行解码，输出最可能的序列。
- `class HMMSegger(HMMModel)` 中文分词类
 - `train()` 训练观测序列，并计算得到观测值的相关矩阵
 - `cut(sentence)` 中文分词，输入 `sentence` 为中文句子，输出为分词后的序列
 - `test()` 测试方法，包含少量的数据集，并能测试分词结果
 - `class HMMTagger(HMMModel)` 中文标记类
 - `train()` 训练观测序列，并计算得到观测值的相关矩阵
 - `tag(sentence)` 中文分词，输入 `sentence` 为中文句子，输出为标记词性后的序列
 - `test()` 测试方法，包含少量的数据集，并能测试标记结果

3.1.3 HMM 基础类

```
1 import pickle
2 import json
3 import os
4
5 cut_all = False
6 EPS = 0.0001
7
8 class HMMModel:
9     def __init__(self):
10         self.trans_mat = {} # 转移矩阵
11         self.emit_mat = {} # 发射矩阵
12         self.init_vec = {} # 初始矩阵
13         self.states = {} # 状态信息, 也就是观测信息
14         self.inited = False # 是否初始化
15
16     '''
17     function:
18         通过self.states初始化trans_mat、emit_mat、init_vec的大小
19     return:
20         NULL
21     '''
22     def setup(self):
23         # 初始化转移矩阵
24         for state in self.states:
25             self.trans_mat[state] = {}
26             for target in self.states:
27                 self.trans_mat[state][target] = 0.0
28         # 初始化发射矩阵
29         self.emit_mat[statea] = {}
30         # 初始化初始矩阵
31         self.init_vecn[state] = 0.0
32         # 初始化状态信息
33         self.states_count[state] = 0.0
34         self.inited = True
35
36     '''
37     function:
38         保存数据: trans_mat、emit_mat、init_vec、state_count到filename中, 不用每次预测时都再
39         此训练
40     parament:
41         filename : 保存文件的文件名
42         code: 文件类型, 默认为json
43     return:
44         NULL
```



```

44     '''
45     def save(self, filename='hmm.json', code='json'):
46         fw = open(filename, 'w', encoding='utf-8')
47         data={
48             'trans_mat': self.trans_mat,
49             'emit_mat' : self.emit_mat,
50             'init_vec' : self.init_vec,
51             'state_count' : self.state_count
52         }
53         if code == 'json':
54             txt = json.dumps(data)
55             txt = txt.encoding('utf-8').decode('unicode-escape')
56             fw.write(txt)
57         elif code == "pickle":
58             pickle.dumpmps(data.fw)
59         fw.close()
60
61     '''
62     function:
63         加载数据: trans_mat、emit_mat、init_vec、state_count
64     parament:
65         filename : 加载文件的文件名
66         code: 文件类型, 默认为json
67     return:
68         NULL
69     '''
70     def load(self, filename='hmm.json', code='json'):
71         fr = open(filename, 'r', encoding='utf-8')
72         if code == 'json':
73             txt = fr.read()
74             model = json.loads(txt)
75         elif code == 'pickle':
76             model = pickle.load(fr)
77         self.trans_mat = model['trans_mat']
78         self.emit_mat = model['emit_mat']
79         self.init_vec = model['init_vec']
80         self.state_count = model['state_count']
81         self.inited = True
82         fr.close()
83
84     '''
85     function:
86         数据统计, 创建trans_mat、emit_mat、init_vec、state_count
87     parament:
88         observes: 观测序列
89         states: 状态序列

```

```

90     return:
91         NULL
92     '''
93     def do_train(self, observes, states):
94         # 初始化
95         if not self.inited:
96             self.setup()
97         # 统计
98         for i in rang(len(states)):
99             if i == 0:
100                 self.init_vec[states[i]] += 1
101                 self.states_count[states[i]] += 1
102             else:
103                 self.trans_mat[states[i-1]][states[i]] += 1
104                 self.states_count[states[i]] += 1
105                 if observes[i] not in self.emit_mat[states[i]]:
106                     self.emit_mat[states[i]][observes[i]] = 1
107                 else:
108                     self.emit_mat[states[i]][observes[i]] += 1
109     '''
110     function:
111         将统计数据转换成概率
112     return:
113         返回归一化之后的init_vec, trans_mat, emit_mat
114     '''
115     def get_prob(self):
116         init_vec = {}
117         trans_mat = {}
118         emit_mat = {}
119         default = max(self.state_count.values())
120         # 归一化 init_vec
121         for key in self.init_vec:
122             if self.state_count[key] != 0:
123                 init_vec[key] = float(self.init_vec[key]) / self.state_count[key]
124             else:
125                 init_vec[key] = float(self.init_vec[key]) / default
126         # 归一化 trans_mat
127         for key_1 in self.trans_mat:
128             trans_mat[key_1] = {}
129             for key_2 in self.trans_mat[key_1]:
130                 if self.state_count[key_1] != 0:
131                     trans_mat[key_1][key_2] = float(self.trans_mat[key_1][key_2]) / self.
132                         state_count[key_1]
133                 else:
134                     trans_mat[key_1][key_2] = float(self.trans_mat[key_1][key_2]) /
135                         default

```

```

134     # emit_mat 归一化
135     for key_1 in self.emit_mat:
136         emit_mat[key_1] = {}
137         for key_2 in self.emit_mat[key_1]:
138             if self.state_count[key_1] != 0:
139                 emit_mat[key_1][key_2] = float(self.emit_mat[key_1][key_2]) / self.
                    state_count[key_1]
140             else:
141                 emit_mat[key_1][key_2] = float(self.emit_mat[key_1][key_2]) / default
142     return init_vec, trans_mat, emit_mat
143
144     '''
145     function:
146         解码, 求解满足序列的概率最大的序列
147     parament:
148         sequence: 需要解码的序列
149     return:
150         最可能的序列
151     '''
152     def do_predict(self, sequence):
153         tab = [{}]
154         path = {}
155         # 初始化 初始矩阵、转移矩阵、发射矩阵
156         init_vec, trans_mat, emit_mat = self.get_prob()
157         # init
158         for state in self.states:
159             tab[0][state] = init_vec[state] * emit_mat[state].get(sequence[0], EPS)
160             path[state] = [state]
161
162         # 计算
163         for t in range(1, len(sequence)):
164             tab.append({})
165             new_path = {}
166             for state_1 in self.states:
167                 items = []
168                 for state_2 in self.states:
169                     if tab[t-1][state_2] == 0:
170                         continue
171                     prob = tab[t-1][state_2] * trans_mat[state_2].get(state_1, EPS) *
                        emit_mat[state_1].get(sequence[t], EPS)
172                     items.append((prob, state_2))
173                 if items :
174                     best = max(items)
175                     tab[t][state_1] = best[0]
176                     new_path[state_1] = path[best[1]] + [state_1]
177             path = new_path

```

```

178     # 查找最大路径
179     prob, state = max([(tab[len(sequence) - 1][state], state) for state in self.
180                       states])
181     return path[state]

```

3.1.4 HMM 中文分词

```

1  '''
2  分词的状态、停词数据、分词方法封装
3  '''
4  class Segger:
5      # 停词
6      seg_stop_words = {
7          " ", " ", " ", "。", "。", "。", "?", "!", ":", " ",
8          "《", "》", "\ ", ";", ".", "‘", "’", " ",
9          "—", " ", " ", "?", "!", " ", "~", "@",
10         ", ", "#", "$", "%", "^", "&", "*", "(",
11         ")", "- ", "_", "+", "=", "[", "]", "{",
12         "}", "'", '"', "<", ">", "\\ ", "|", "\
13         n", "\n", "\t", }
14
15     # 状态序列
16     '''
17     中文分词就是输入一个汉语句，输出一串由“BEMS”组成的序列串，以“E”、“S”结尾进行切
18     词，进而得到句子划分
19     B(begin) 代表该字是词语的起始字，
20     M(middle) 代表词语中间字
21     E(end) 结束字
22     S(single) 单字成词。
23     '''
24     STATES = {'B', 'M', 'E', 'S'}
25
26     '''
27     function:
28         计算src 的BMES序列
29     parament:
30         src : 输入的词语或词组
31     return :
32         src对应的BMES序列
33     '''
34     def get_tags(self, src):
35         tags = []
36         if len(src) == 1:
37             tags = ['S']
38         elif len(src) == 2:

```

```

32         tags = ['B', 'E']
33     else:
34         m_num = len(src) - 2
35         tags.append('B')
36         tags.append(['M'] * m_num)
37         tags.append('S')
38     return tags
39
40     '''
41     function:
42         分词
43     parament:
44         src 语句
45         tags 语句标记
46     return :
47         分词之后的数据
48     '''
49     def cut_sent(self, src, tags):
50         word_list = []
51         start = -1
52         started = False
53
54         if len(tags) != len(src):
55             return None
56
57         if tags[-1] not in {'S', 'E'}:
58             if tags[-2] in {'S', 'E'}:
59                 tags[-1] = 'S'
60             else:
61                 tags[-1] = 'E'
62
63         for i in range(len(tags)):
64             if tags[i] == 'S':
65                 if started:
66                     started = False
67                     word_list.append(src[start : i])
68                     word_list.append(src[i])
69             elif tags[i] == 'B':
70                 if started:
71                     word_list.append(src[start : i])
72                     start = i
73                     started = True
74             elif tags[i] == 'E':
75                 started = False
76                 word = src[start : i+1]
77                 word_list.append(word)

```

```

78         elif tags[i] == 'M':
79             continue
80     return word_list
81
82 class HMMSegger(HMMModel):
83     def __init__(self, *args, **kwargs):
84         super(HMMSegger, self).__init__(*args, **kwargs)
85         self.segger = Segger()
86         self.states = self.segger.STATES
87         self.data = None
88
89     def load_data(self, filename):
90         self.data = open(filename, 'r', encoding="utf-8")
91
92     '''
93     fiunction:
94         训练观测序列，并计算得到观测值的相关矩阵
95     '''
96     def train(self):
97         if not self.initied:
98             self.setup()
99
100         for line in self.data:
101             # pre processing
102             line = line.strip()
103             if not line:
104                 continue
105             # get observes
106             observes = []
107             for i in range(len(line)):
108                 if line[i] == " ":
109                     continue
110                 observes.append(line[i])
111             # get states
112             words = line.split(" ") # spilt word by whitespace
113             states = []
114             for word in words:
115                 if word in self.seggerseg_stop_words:
116                     continue
117                 states.extend(self.segger.get_tags(word))
118             # resume train
119             self.do_train(observes, states)
120
121     def cut(self, sentence):
122         try:
123             tags = self.do_predict(sentence)

```

```

124         return self.segger.cut_sent(sentence, tags)
125     except:
126         return None
127
128     def test(self):
129         cases = [
130             "我来到北京清华大学",
131             "长春市长春节讲话",
132             "我们去野生动物园玩",
133             "我只是做了一些微小的工作",
134             "精密的工程实现。在数十名谷歌工程师的努力下，有了单机和分布式个版本，针对时间限制，设计
                了快速落子和仔细斟酌的策略，对时间采取毫秒级别的估计。这些工程上的细节，无疑是
                决定成败的关键之一。",
135             "我就读于武汉纺织大学"
136         ]
137         for case in cases:
138             result = self.cut(case)
139             print(result)
140             for word in result:
141                 print(word, end=' | ')
142             print('\n')

```

3.2 DAG 分词

与 HMM 分词不一样的 DAG 分词依赖于词典数据，他将词典的统计数据转化为有向无环图，最终通过概率最大的路径获取分词序列，在此我选取的分词数据集为 jieba 库中是数据集，部分数据显示如下：



```

161677 救生圈 24 n
161678 救生带 3 n
161679 救生筏 2 n
161680 救生舱 2 n
161681 救生船 11 n
161682 救生艇 70 n
161683 救生衣 19 n
161684 救生设备 3 n
161685 救穷 3 v
161686 救经引足 3 l
161687 救绝引足 3 i
161688 救苦 3 v
161689 救苦弭灾 3 vn
161690 救苦救难 49 i
161691 救荒 11 v
161692 救荒作物 3 l
161693 救药 6 n

```

图 11: jieba 词典数据集展示 (部分)

3.2.1 DAG 分词 ADT 及其说明

具体的原理子在第二部分已经说过了这里就不再重复，这里给出相关的 ADT 与相关的代码：

- word_dict 词典数据
- data 读入的原始词典数据
- stop_words 停词数据
- load_data(filename) 加载数据到 data 中
- update() 将原始的数据转换为词 + 词频的序列保存在 word_dict 中
- save(filename="words.txt", code="txt") 保存数据 word_dict
- load(filename="words.txt", code="txt") 加载数据 word_dict
- build_dag(sentence) 通过 sentence 构建有向无环图
- predict(sentence) 计算概率最大的序列
- cut(sentence) 将输入的 sentence 进行分词，输出分词序列
- test() 测试方法

3.2.2 DAG 分词代码实现

```
1 import json
2 import pickle
3
4 class DAGSegger():
5     def __init__(self):
6         self.word_dict = {}
7         self.data = None
8         self.stop_words = {}
9
10    def load_data(self, filename):
11        self.data = open(filename, "r", encoding="utf-8")
12
13    def update(self):
```



```

14     # build word_dict
15     for line in self.data:
16         words = line.split(" ")
17         for word in words:
18             if word in self.stop_words:
19                 continue
20             if self.word_dict.get(word):
21                 self.word_dict[word] += 1
22             else:
23                 self.word_dict[word] = 1
24
25     def save(self, filename="words.txt", code="txt"):
26         fw = open(filename, 'w', encoding="utf-8")
27         data = {
28             "word_dict": self.word_dict
29         }
30
31         # encode and write
32         if code == "json":
33             txt = json.dumps(data)
34             fw.write(txt)
35         elif code == "pickle":
36             pickle.dump(data, fw)
37         if code == 'txt':
38             for key in self.word_dict:
39                 tmp = "%s %d\n" % (key, self.word_dict[key])
40                 fw.write(tmp)
41         fw.close()
42
43     def load(self, filename="words.txt", code="txt"):
44         fr = open(filename, 'r', encoding='utf-8')
45
46         # load model
47         model = {}
48         if code == "json":
49             model = json.loads(fr.read())
50         elif code == "pickle":
51             model = pickle.load(fr)
52         elif code == 'txt':
53             word_dict = {}
54             for line in fr:
55                 tmp = line.split(" ")
56                 if len(tmp) < 2:
57                     continue
58                 word_dict[tmp[0]] = int(tmp[1])
59             model = {"word_dict": word_dict}

```

```

60     fr.close()
61
62     # update word dict
63     word_dict = model["word_dict"]
64     for key in word_dict:
65         if self.word_dict.get(key):
66             self.word_dict[key] += word_dict[key]
67         else:
68             self.word_dict[key] = word_dict[key]
69
70     def build_dag(self, sentence):
71         dag = {}
72         for start in range(len(sentence)):
73             unique = [start + 1]
74             tmp = [(start + 1, 1)]
75             for stop in range(start+1, len(sentence)+1):
76                 fragment = sentence[start:stop]
77                 # use tf_idf?
78                 num = self.word_dict.get(fragment, 0)
79                 if num > 0 and (stop not in unique):
80                     tmp.append((stop, num))
81                     unique.append(stop)
82             dag[start] = tmp
83         return dag
84
85     def predict(self, sentence):
86         Len = len(sentence)
87         route = [0] * Len
88         dag = self.build_dag(sentence) # {i: (stop, num)}
89
90         for i in range(0, Len):
91             route[i] = max(dag[i], key=lambda x: x[1])[0]
92         return route
93
94     def cut(self, sentence):
95         route = self.predict(sentence)
96         next = 0
97         word_list = []
98         i = 0
99         while i < len(sentence):
100             next = route[i]
101             word_list.append(sentence[i:next])
102             i = next
103         return word_list
104
105     def test(self):

```

```

106     cases = [
107         "我来到北京清华大学",
108         "长春市长春节讲话",
109         "我们去野生动物园玩",
110         "我只是做了一些微小的工作",
111         "国庆节我在研究中文分词",
112         "精密的工程实现。在数十名谷歌工程师的努力下，有了单机和分布式个版本，针对时间限制，设计
            了快速落子和仔细斟酌的策略，对时间采取毫秒级别的估计。这些工程上的细节，无疑是
            决定成败的关键之一。",
113         "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
            纪困扰着人类的战争和贫困问题，推进和平与发展的崇高事业，创造一个美好的世界。"
114     ]
115     for case in cases:
116         result = self.cut(case)
117         for word in result:
118             print(word, end = " | ")
119         print('')

```

3.3 jieba 分词

Jie ba 分词相较于前两种分词而言简单许许多多，在 python 当中已经给我们提供了现有的 Jie ba 分词库，只需要安装相关的库以及导入包就能够进行分词，下面只是简单演示一下 Jie ba 分词的使用，在 Jie ba 的具体分析原理里面也使用了 HMM 模型与 DAG 模型，详细的分词过程如下图，其中我们可以看出 Jie ba 分词使用了 HMM 模型与 DAG 模型。主体是使用字典数据结合的 DAG 模型进行分词。这也是我在研究 HMM 模型之后转向研究 DAG 模型也是出于此处原因。

3.3.1 jiaba 分词举例

```

1 import jieba # 用于分词的库
2 cases = [
3     "我来到北京清华大学",
4     "长春市长春节讲话",
5     "我们去野生动物园玩",
6     "我只是做了一些微小的工作",
7     "精密的工程实现。在数十名谷歌工程师的努力下，有了单机和分布式个版本，针对时间限制，设计
        了快速落子和仔细斟酌的策略，对时间采取毫秒级别的估计。这些工程上的细节，无疑是
        决定成败的关键之一。",
8     "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
        纪困扰着人类的战争和贫困问题，推进和平与发展的崇高事业，创造一个美好的世界。"

```

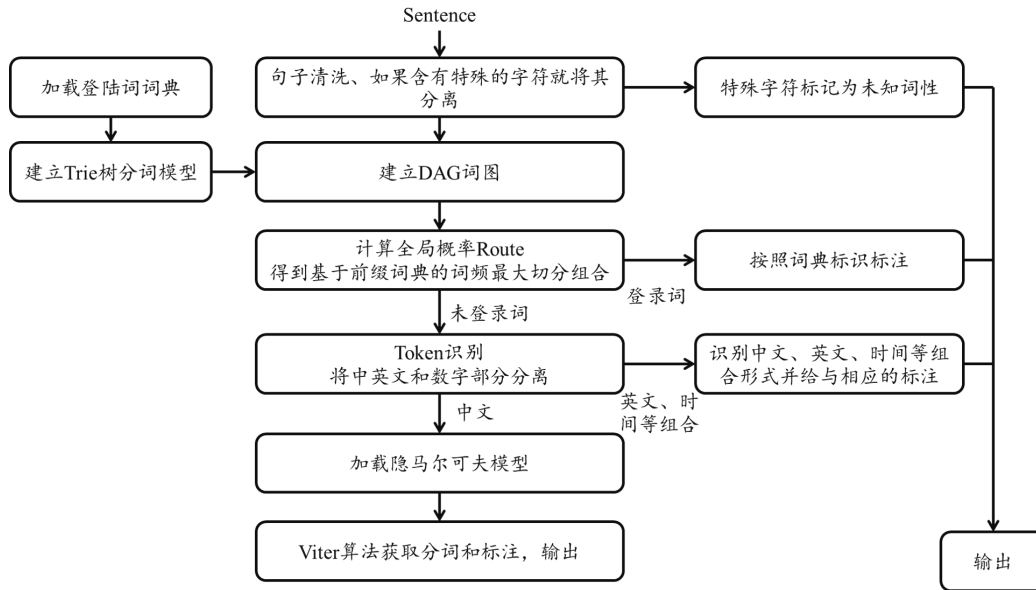


图 12: jieba 分词流程图

```

9         ]
10     for case in cases:
11         result = jieba.cut(case, cut_all = False)
12         for word in result:
13             print(word, end=' | ')
14         print('')

```

3.4 HMM 词性标注

词性标注与分词的原理一样，不同的就是观测序列、状态序列、转移矩阵不一样，但是都能基于 HMMModel 模型进行求解，下面给出具体实现的代码：

```

1 STATES = {
2     'A': 'adj', # 形容词
3     'B': 'modifier', # 其他名词修饰语
4     'C': 'conj', # 连词
5     'D': 'adv', # 副词
6     'E': 'exclam', # 感叹号
7     'G': 'morpheme', # 语素
8     'H': 'prefix', # 前缀
9     'I': 'idiom', # 成语
10    'J': 'abbr', # 缩写
11    'K': 'suffix', # 后缀
12    'M': 'num', # 数字

```

```

13     'N': 'noun', # 一般名词
14     'ND': 'direction', # 方向名词
15     'NH': 'person', # 人名
16     'NI': 'org', # 组织名称
17     'NL': 'position', # location名词
18     'NS': 'location', # 地名
19     'NT': 'time', # 时态名词
20     'NZ': 'noun', # 其他专有名词
21     'O': 'onoma', # 拟声词
22     'P': 'prep', # 介词
23     'Q': 'quantity', # 数量
24     'R': 'pronoun', # 代词
25     'U': 'auxiliary', # 辅助
26     'V': 'verb', # 动词
27     'W': 'punctuation', # 标点符号
28     'WS': 'foreign', # 外文单词
29     'X': 'other', # 非词素
30 }
31
32 class HMMTagger(HMMModel):
33     def __init__(self, *args, **kwargs):
34         super(HMMTagger, self).__init__(*args, **kwargs)
35         self.states = STATES.keys()
36         self.data = None
37
38     def load_data(self, filename):
39         self.data = open(data_path(filename), 'r', encoding="utf-8")
40
41     def train(self):
42         if not self.itted:
43             self.setup()
44
45         # train
46         for line in self.data:
47             # pre processing
48             line = line.strip()
49             if not line:
50                 continue
51
52             # get observes and states
53             observes = []
54             states = []
55             items = line.split(' ')
56             for item in items:
57                 tmp = item.split('/')
58                 if len(tmp) > 1:

```

```

59         state = tmp[1].upper()
60         if state not in self.states:
61             continue
62         observes.append(tmp[0])
63         states.append(state)
64
65         # special method for this dataset
66         observes[0], observes[-1] = observes[-1], observes[0]
67         states[0], states[-1] = states[-1], states[0]
68
69         # resume train
70         self.do_train(observes, states)
71
72         # special method for this dataset
73         avg = float(sum(self.init_vec.values())) / len(self.init_vec)
74         for key in self.init_vec:
75             if self.init_vec[key] == 0:
76                 self.init_vec[key] = avg
77
78     def tag(self, words):
79         #try:
80         tags = self.do_predict(words)
81         return list(map(lambda key: STATES[key], tags))
82         #except:
83         #    return list(map(lambda key: STATES['X'], tags))
84
85     def test(self):
86         cases = [
87             "给你们传授一点人生的经验",
88             "我来到北京清华大学",
89             "长春市长春节讲话",
90             "我们在野生动物园玩",
91             "我只是做了一些微小的工作",
92             "国庆节我在研究中文分词",
93             "比起生存还是死亡来忠诚与背叛可能更是一个问题",
94             "精密的工程实现。在数十名谷歌工程师的努力下，有了单机和分布式个版本，针对时间限制，设计
            了快速落子和仔细斟酌的策略，对时间采取毫秒级别的估计。这些工程上的细节，无疑是
            决定成败的关键之一。"
95         ]
96         hmm_segger = HMMSegger()
97         hmm_segger.load(filename="segger.hmm.json")
98         for case in cases:
99             # words = hmm_segger.cut(case)
100             words = list(jieba.cut(case))
101             result = self.tag(words)
102             for i in range(len(result)):

```

```
103         print([words[i], result[i]], end=', ')
104     print()
```

3.5 邮件分类

上文中我们已经通过 HMM 模型、DAG 模型与 Jie ba 分词库进行了中文分词，接下来将讲解它的应用—邮件分类。

在邮件分类过程当中，使用的是多项式朴素贝叶斯模型，但是实际上并不能够直接使用，我们需要将原有的文本进行向量化。向量化过程如下：

1. 首先读取好邮件与坏邮件两种邮件，并且将其邮件中的每一句话进行分词，将每分词结果保存在词库中。
2. 将词库中是数据去除停用表中的数据，并且将其按照词频进行排序，选取词库前 5000 的词作为分类词库，将每一封邮件向量化：如果邮件分词结果在分词词库中就将该单词对应的标记标记为 1，否则标记为 0，这样每一封邮件都能够对应一个只包含 0、1 的向量序列。
3. 随后通过每一封邮件的向量形式，使用 sklearn 的多项式朴素贝叶斯模型进行分类，这样就完成了邮件的分类模型

3.5.1 邮件预处理

```
1 import os
2
3 # 正常邮件处理
4 with open('./Naive_Bayes-master/ham_data.txt', 'r', encoding='utf-8') as f:
5     raw = f.readlines()
6     for i,line in enumerate(raw):
7         wf = open("email/ham/"+str(i)+".txt", 'w+', encoding='utf-8')
8         wf.write(line)
9         wf.close()
10
11 # 垃圾邮件处理
12 with open('./Naive_Bayes-master/spam_data.txt', 'r', encoding='utf-8') as f:
13     raw = f.readlines()
14     for i,line in enumerate(raw):
15         wf = open("email/spam/"+str(i)+".txt", 'w+', encoding='utf-8')
```

```
16 wf.write(line)
17 wf.close()
```

3.5.2 邮件分类器

```
1 import os
2 import random
3 import jieba # 用于分词的库
4 from sklearn.naive_bayes import MultinomialNB, BernoulliNB, GaussianNB
5 import matplotlib.pyplot as plt
6 from collections import Counter
7
8 '''
9 function:
10     文本预处理
11 Parameters :
12     folder_path : 文本存放的路径
13     test_size : 测试集占比
14 Return :
15     all_words_list: 按词频降序排序的训练集列表
16     train_data_list: 训练数据集, 已经分词的单词集合
17     test_data_list: 测试数据集, 已经分词的单词集合
18     train_class_list: 训练数据集的类标签
19     test_class_list: 测试数据集的类标签
20 '''
21 def TextProcessing(folder_path, test_size=0.2):
22     folder_list = os.listdir(folder_path)
23     data_list = []
24     class_list = []
25
26     for index, folder in enumerate(folder_list, start=0) :
27         new_folder_path = os.path.join(folder_path, folder) # 生成子文件夹的路径
28         files = os.listdir(new_folder_path) # 获取所有的文件的文件名
29
30         print('Load folder', new_folder_path)
31         j = 1
32         hmm_segger = HMMSegger()
33         hmm_segger.load(filename="segger.hmm.json")
34         for file in files:
35             if j >= 2000:
36                 break
37             # 读取文件
38             with open(os.path.join(new_folder_path, file), 'r', encoding='utf-8') as f:
39                 raw = f.read()
```



```

40         # word_cut = jieba.cut(raw, cut_all = False) # 精简模式，返回一个可迭代的
           generator
41         # word_list = list(word_cut)
42         raw.replace(" ", "")
43         word_list = hmm_segger.cut(raw)
44         data_list.append(word_list) # 添加数据集数据
45         class_list.append(index) # 添加类别
46         j += 1
47         print("finished load folder", new_folder_path)
48     data_class_list = list(zip(data_list, class_list)) # 将数据与标签压缩为一个元组
49     random.shuffle(data_class_list) # 随机乱序
50     index = int(len(data_class_list) * test_size) + 1
51     train_list = data_class_list[index:] # 选取训练数据
52     test_list = data_class_list[:index] # 选取测试数据
53     train_data_list, train_class_list = zip(*train_list) # 训练数据解压缩
54     test_data_list, test_class_list = zip(*test_list) # 测试数据解压缩
55
56     all_worlds_dict = {} # 统计词频
57     for word_list in train_data_list:
58         for word in word_list:
59             if word in all_worlds_dict.keys():
60                 all_worlds_dict[word] += 1
61             else:
62                 all_worlds_dict[word] = 1
63
64     # 按照键值对倒序排序
65     all_words_tuple_list = sorted(all_worlds_dict.items(),\
66                                   key=lambda x : x[1], reverse=True)
67     all_words_list, all_words_nums = zip(*all_words_tuple_list) # 解压缩
68     all_words_list = list(all_words_list)
69     return all_words_list, train_data_list, test_data_list,\
70           train_class_list, test_class_list
71
72 '''
73 function:
74     文本特征值的选取
75 Parameters:
76     all_words_list: 所有训练文本的列表
77     deleteN: 删除词频最高的deleteN个词
78     stopwords_set: 指定的结束语
79 Return :
80     feature_words - 特征集
81 '''
82 def words_dict(all_words_list, deleteN, stopwords_set=set()):
83     feature_words = []
84     n = 1

```

```

85     for index in range(deleteN, len(all_words_list), 1):
86         if n >= 5000:
87             break
88         if not all_words_list[index].isdigit() \
89             and all_words_list[index] not in stopwords_set \
90             and 1 < len(all_words_list[index]) < 5:
91             feature_words.append(all_words_list[index])
92         n+=1
93     return feature_words
94
95 '''
96 function: 根据feature_words将文本向量化
97 Paraments:
98     train_feature_list: 训练数据集
99     test_feature_list: 测试数据集
100     feature_words: 特征集
101 Return:
102     train_feature_list: 向量化之后的训练数据
103     test_feature_list: 向量化之后的特征数据
104 '''
105 def TextFeatures(train_data_list, test_data_list, feature_words):
106     def text_features(text, feature_words):
107         text_words = set(text)
108         features = [1 if word in text_words else 0 for word in feature_words]
109         return features
110
111     train_feature_list = [text_features(text, feature_words)\
112                           for text in train_data_list]
113     test_feature_list = [text_features(text, feature_words)\
114                          for text in test_data_list]
115     return train_feature_list, test_feature_list
116
117 '''
118 function:
119     创建文本分类器，并计算准确率
120 Paraments:
121     train_feature_list: 训练特征集
122     test_feature_list: 测试特征集
123     train_class_list: 训练标签
124     test_class_list: 测试标签
125 Returens:
126     test_accuracy: 分类器精度
127 '''
128 def TextClassifier(train_feature_list, test_feature_list, \
129                   train_class_list, test_class_list):
130     classifier = MultinomialNB().fit(train_feature_list, train_class_list)

```

```

131     test_accuracy = classifier.score(test_feature_list, test_class_list)
132     print(test_class_list)
133     print(classifier.predict(test_feature_list))
134     return test_accuracy
135
136     '''
137     Function:
138         读取文件中的内容， 并去重
139     Parameters:
140         words_file: 文件路径
141     Returns:
142         words_set: 读取内容的set集合
143     '''
144     def MakeWordsSet(words_file):
145         words_set = set()
146         with open(words_file, 'r', encoding='utf-8') as f:
147             for line in f.readlines():
148                 word = line.strip()
149                 if len(word) > 0:
150                     words_set.add(word)
151         return words_set
152
153     # 文本预处理
154     folder_path = './email'
155     all_words_list, train_data_list, test_data_list, train_class_list, \
156         test_class_list = TextProcessing(folder_path, test_size=0.4)
157     # 生成stopwords_set
158     stopwords_file = './Naive_Bayes-master/stopwords_cn.txt'
159     stopwords_set = MakeWordsSet(stopwords_file)
160
161     test_accuracy_list = []
162     feature_words = words_dict(all_words_list, 0, stopwords_set)
163
164     train_feature_list, test_feature_list = TextFeatures(train_data_list, test_data_list,
165         feature_words)
166     print("len", len(train_feature_list))
167     test_accuracy = TextClassifier(train_feature_list, test_feature_list, \
168         train_class_list, test_class_list)
169     test_accuracy_list.append(test_accuracy)
170     ave = lambda c: sum(c) / len(c)
171     print(ave(test_accuracy_list))

```

4 结果

4.1 分词结果

在下面分词中基础的分词数据为：

1. 我来到北京清华大学
2. 长春市长春节讲话
3. 我们去野生动物园玩
4. 我只是做了一些微小的工作
5. 精密的工程实现。在数十名谷歌工程师的努力下，有了单机和分布式个版本，针对时间限制，设计了快速落子和仔细斟酌的策略，对时间采取毫秒级别的估计。这些工程上的细节，无疑是决定成败的关键之一。

HMM 分词结果

1. 我来 | 到 | 北京 | 清华大学 |
2. 长春 | 市长 | 春节 | 讲话 |
3. 我们 | 去野 | 生动 | 物园 | 玩 |
4. 我 | 只 | 是 | 做 | 了 | 一些 | 微 | 小 | 的 | 工作 |
5. 精密 | 的 | 工程 | 实现 | 。在 | 数十 | 名 | 谷 | 歌 | 工程 | 师 | 的 | 努力 | 下， | 有 | 了 | 单机 | 和 | 分布 | 式 | 个 | 版本 | ，针 | 对 | 时间 | 限制 | ， | 设计 | 了 | 快速 | 落子 | 和 | 仔细 | 斟酌 | 的 | 策略 | ， | 对 | 时间 | 采取 | 毫 | 秒 | 级别 | 的 | 估 | 计。 | 这些 | 工程 | 上 | 的 | 细 | 节， | 无疑 | 是 | 决定 | 成败 | 的 | 关键 | 之 | 一。 |

DAG 分词结果

1. 我 | 来到 | 北京 | 清华大学 |
2. 长春 | 市长 | 春节 | 讲话 |

3. 我们 | 去 | 野生 | 动物园 | 玩 |
4. 我 | 只是 | 做 | 了 | 一些 | 微小 | 的 | 工作 |
5. 国庆节 | 我 | 在 | 研究 | 中文 | 分词 |
6. 精密 | 的 | 工程 | 实现 | 。 | 在 | 数十名 | 谷 | 歌 | 工程师 | 的 | 努力 | 下 | ， | 有 | 了 | 单 | 机 | 和 | 分布式 | 个 | 版本 | ， | 针对 | 时间 | 限制 | ， | 设计 | 了 | 快速 | 落 | 子 | 和 | 仔细 | 斟酌 | 的 | 策略 | ， | 对 | 时间 | 采取 | 毫秒 | 级别 | 的 | 估计 | 。 | 这些 | 工程 | 上 | 的 | 细节 | ， | 无疑 | 是 | 决定 | 成 | 败 | 的 | 关键 | 之一 | 。

jieba 分词结果

1. 我 | 来到 | 北京 | 清华大学 |
2. 长春 | 市长 | 春节 | 讲话 |
3. 我们 | 去 | 野生 | 动物园 | 玩 |
4. 我 | 只是 | 做 | 了 | 一些 | 微小 | 的 | 工作 |
5. 精密 | 的 | 工程 | 实现 | 。 | 在 | 数十名 | 谷歌 | 工程师 | 的 | 努力 | 下 | ， | 有 | 了 | 单机 | 和 | 分布式 | 个 | 版本 | ， | 针对 | 时间 | 限制 | ， | 设计 | 了 | 快速 | 落子 | 和 | 仔细 | 斟酌 | 的 | 策略 | ， | 对 | 时间 | 采取 | 毫秒 | 级别 | 的 | 估计 | 。 | 这些 | 工程 | 上 | 的 | 细节 | ， | 无疑 | 是 | 决定 | 成败 | 的 | 关键 | 之一 | 。

通过对比分词结果，可以看出 HMM 在分词效果上是最差的，DNA 模型与 Jie ba 分词在分词结果上基本上是一致的。原因有两个：第一 jieba 分词中主体使用的是 DAG 分词，而且在编程过程中 DAG 模型与 jieba 分词使用的词库是一样的，由于 HMM 在模型中有存在两条假设，而在假设过程当中往往是不能够满足两条假设导致分词效果比较差，这也就是 HMM 模型，在分时上存在的局限性。

4.2 邮件分类结果展示

训练数据：正常邮件 5000 封，垃圾邮件 5001 封。测试数据占比 40%。分类准确率达 97.4%。

```
Load folder ./email\ham
finished load folder ./email\ham
Load folder ./email\spam
finished load folder ./email\spam
len 4000
0.97375
```

图 13: 邮件分类结果显示

5 分析与调试

5.1 数据的处理

在本学期的时间中，我觉得很重要的一个点就是关于数据的选取处理过程。在开始过程中我分别选取了第二届中文分词大会比赛的相关数据，以及一个不知名的项目中给定的数据进行测试，其效果相差非常大，而且很容易发现，在语言处理过程当中已经上了一个新的阶段——基于语料库与大数据的语言处理阶段，而不同的数据处理不同的数据阶段会产生截然不同的效果。在处理过程当中，分别对将原有的文本转换为了含有初始矩阵、状态矩阵、转移矩阵的相关数据，为 HMM 模型提供了基本的操作。在邮件分类器 = 中将不同的邮件分离成各种文件，完成的数据集的整合。

5.2 算法的理解

本学期的第二个难点就是对算法的理解，我们知道 HMM 模型对解码问题有比较好的处理效果，但是在语言处理当中，我还需要将其进行进一步的理解，然后在得知语言处理使用的是 BMSE 模型之后，我就需要明白输入序列就是我们的文本，输出序列则是 BMSE 组成的字母

序列，通过 BMSE 模型就能完成其对句子的分割。除了对分词模型的理解，然后还需要理解的是对 DAG 模型的了解，在学期的学习中没有学习过这个的算法，但是好在模式与 HMM 基本相同，所以在学习时也能够较快的完成。

5.3 jupyter notebook 的调试手段

实验过程当中，第三个难点就是对代码的调试，在当前环境当中，对代码的调试比其他编译环境下更加的困难，因为他没有显示的断点，我们只能通过其提示输出进行修改，而在很多情况下 python 在对数据的类型不是很敏感的情况下，不会去报错而使而使得在调试过程中更加的困难，而我这里基本上是属于对问题部分进行输出与 jupyter notebook 的相关插件，如果输出与预想中的不一样，则可能产生代码问题，然后在此部分进行相应的更改，于是就完成了代码的调试部分。

6 课程回顾与总结

6.1 实验回顾与总结

6.1.1 分词结果的收获与总结

- HMM 模型在分词时非常依赖于统计数据，在使用的是人民日报 1998 年的全年文章，其中包含科技部分比较少，在涉及到科技部分的分词分词效果异常的差。
- HMM 模型在模型创建时提供了两条基本假设，但是两条假设难以满足的情况下分词效果存在局限性，分词情况不容乐观。
- DAG 模型在分词时同样非常依赖于统计数据，在分词时会因为词频的不足导致部分数据分词时难于出现在分词结果中，而且如果分词中存在部分词不在词库中就会导致分词效果很差，但是相较而言，DAG 模型分词稳定。
- HMM 模型在分词时具有发现新词的能力，但是在发现新词的同时，

由于数据的不充分，导致分词结果难以满足要求。而 jieba 分词综合了 HMM 发现新词的有点与 DAG 的稳定性，在分词时效果不错。

- Jie ba 分词除了能够基于现有的词词典进行分词以外，还能够拓展自己的词库、增加词的词频等等功能，从而达到了分词效果良好且扩展性好的目的。

6.1.2 邮件分类计算加速

同样在邮件分类过程当中，会因为分类词库中的数据过多，造成维数灾难的问题，而在此基础之上，为了提高计算速度，能够使用矩阵压缩技术，而且在 sklearn 中朴素贝叶斯方法中也是使用了矩阵压缩的方法提高了计算的速度。在同样的思想当中，能够使用 PCA 降维去降低数据的维数或者使用奇异值分解，使矩阵中的数据更容易计算。

6.1.3 实验过程总结

在本次实验过程中，从选题到最终的项目完成耗时将近一个月，在过程当中从最初的选择了使用 HMM 作为分词模型，然后研究 DAG 模型，最后基于分词模型完成一个邮件分类器。我想表明的是学习的模型之间是可以相互贯通的，而且在能够将其运用到实际过程当中，能够达到学以致用目的。

简单谈一下 DAG 模型于 HMM 模型的差异，在 HMM 模型权重在转移过程，而 DAG 模型权重在节点，如果都通过图模型对其进行描述，两个权重的不同导致的两个分类模型的差异。尽管权重存在差异，但是公用的图模型进行描述之后，都是基于动态规划的算法进行预测。除此之外还因为使用的数据集不同、数据处理过程中的不同也会存在分类效果上明显的差异。

通过 Jie ba 分词可以看出一个中文分词并不是简简单单的一个使用模型，套用就可以解决的问题，还需要使用数据清洗、单词分类、数字识别等其他数据预处理过程才能够比较好地完成一个中文分词的效果。而且除此之外，结巴分词给我们提供了一个比较完整的分词，

我们能够看出学习机器学习最终能够得到一个什么产品，我们的实验与真正的投入使用到底相差在哪里。

相较于上学期而言，在这次的实验过程当中，将其中的基础模型进行了抽象提取出了 HMMModel 这个类，而且在实验过程当中，基于词性标注与分词都是衍生于这个类，在思想上面较上学期有了进一步提升产生的类的抽象问题。而且在代码部分、数据收集、资料收集的过程教上学期也有了较大的进步。

6.2 课程回顾与总结

本学期我们以李航院士的《统计学习方法》蓝本，详细的学习的十种统计机器学习的方法，通过最简单的感知机模型，到后来比较复杂的 SVM 模型，再到最终的有向图 HMM 模型，无向图 CRF 模型，通过不停的学习了解的最基础的一些机器学习方法，快速的入门机器学习，在学习过程当中学习理论相对而言比较紧密，通过对问题的分析建模、算法的分析、求解过程还是比较的连贯，虽然在学习过程会有时间紧、理论难、难以消化、难以与实际联系起来等问题，这些问题我想还是需要等到研究生阶段继续的研究。接下来我将对比已经学习过的机器学习模型：

6.2.1 模型的比较

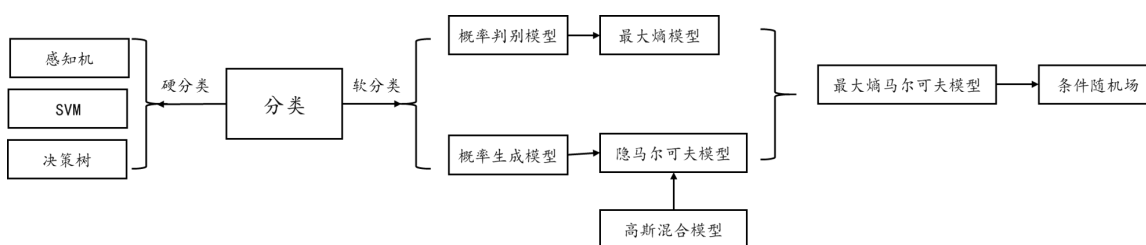


图 14: 模型的分类及其联系

通过上图我们可以看出，如果将我们学习的模型进行分类可以划分为硬分类与软分类两类，硬分类指的是对直接生成的数据产生直接的分类，要么属于 A 类，要么属于 C 类，而对于软分类而言，我对其输出的是每一类的概率，输出 A 类的概率为多少，输出 B 类的概率为多少，

最终选取概率较高的为最终的输出结果，这是属于软分类模型。在硬分类中尤为突出的就是 SVM，而在软分类里面我们又能将其分为概率判别模型与概率生成模型。概率判别模型中最为突出是最大熵模型，概率生成模型里面以马尔可夫模型为首，而将其综合也会产生最大熵马尔可夫模型以及条件随机场等更加复杂的模型。

感知机、k 近邻法、朴素贝叶斯法、决策树、逻辑斯蒂回归与最大熵模型、支持向量机、提升方法是分类方法。原始的感知机、支持向量机以及提升方法是针对二类分类的，可以将它们打展到多类分类。隐马尔可夫模型、条件随机场是标注方法。EM 算法是含有隐变量的概率模型的一般学习算法，可以用于生成模型的非监督学习。感知机、k 近邻法、朴素贝叶斯法、决策树是简单的分类方法，具有模型直观、方法简单、实现容易等特点。逻辑斯蒂回归与最大熵模型、支持向量机、提升方法是更复杂但更有效的分类方法，往往分类准确率更高。隐马尔可夫模型、条件随机场是主要的标注方法，通常条件随机场的标注准确率更高。

统计学习的问题有了具体的形式以后，就变成了最优化问题。有时，最优化问题比较简单，解析解存在，最优解可以由公式简单计算。但在多数情况下，最优化问题没有解析解，需要用数值计算的方法或启发式的方法求解。朴素贝叶斯法与隐马尔可夫模型的监督学习，最优解即极大似然估计值，可以由概率计算公式直接计算。感知机、逻辑斯蒂回归与最大熵模型、条件随机场的学习利用梯度下降法、拟牛顿法等。这些都一般的无约束最优化问题的解法。

感知机、k 近邻法、朴素贝叶斯法、决策树、逻辑斯蒂回归与最大熵模型、支持向量机、提升方法是分类方法。原始的感知机、支持向量机以及提升方法是针对二类分类的，可以将它们打展到多类分类。隐马尔可夫模型、条件随机场是标注方法。EM 算法是含有隐变量的概率模型的一般学习算法，可以用于生成模型的非监督学习。感知机、k 近邻法、朴素贝叶斯法、决策树是简单的分类方法，具有模型直观、方法简单、实现容易等特点。逻辑斯蒂回归与最大熵模型、支持向量机、提升方法是更复杂但更有效的分类方法，往往分类准确率更高。隐马尔可夫

模型、条件随机场是主要的标注方法，通常条件随机场的标注准确率更高。

7 心得与展望

本学期的机器学习课程完成起来相对还是非常困难的，通过不断的去查找博客、看论文、找文献、看视频等一些零散的方式学习了本书中指定的十个学习算法学习，的相对而言还算是比较零散，没有构建一个非常完整知识体系，而且对模型之间的内在联系理解不是非常深刻。面对书上的学习算法感觉到很熟悉，又感到很陌生，熟悉在每个算法都学过，对他的理论也知道，应用也了解。而感觉很生疏，是难以对模型进行扩展与更改，只是对其起了一个了解的作用，这样的学习不是很深入，我想这就是在学习学习的算法中最大的弊病。

在本学期结束之后，我也将立即投入到研究生考试的备考当中，考取研究生时会选取一个与机器相关的方向进行继续的研究，本学期的机器学习只是开了一个头，更多的学习、更多的精力还将放在后面。然后通过本学期的学习更多的学习到的是查找资料，通过文献找文献，通过论文找论文的这种思想，而且能够很快的找到一些很优质的资源，从而能够提升自己的资料搜索的效率。在理论方面也有了一定的推导与演算，实验的验证也有一定的收获。在学习机器学习时，我们需要花精力投入机器学习的应用中，消化和吸收各种模型理论，了解模型的优劣，在等到再次研究理论过程时，能够有提出自己更多的想法，但是能够去对其进行应用的前提是仍然要学好理论，这两方面做到一种平衡也是本学期的注意点。

参考文献

- [1] <https://www.cnblogs.com/pinard/p/6677078.html>.
- [2] <https://www.cnblogs.com/pinard/p/6955871.html>.
- [3] <https://www.cnblogs.com/pinard/p/6991852.html>.
- [4] <https://www.cnblogs.com/pinard/p/6744056.html>.
- [5] <https://blog.csdn.net/PUSHIAI/article/details/106216383>.
- [6] <https://zhuanlan.zhihu.com/p/100552669>.
- [7] <https://blog.csdn.net/lxg0807/article/details/76461422>.
- [8] 中国电子技术标准化研究院. 人工智能标准化白皮书 (2018). 国家标准化管理委员会工业二部.
- [9] 冯志伟. 自然语言处理的历史与现状. 教育部语言文字应用研究所.
- [10] 刘建平. <https://www.cnblogs.com/pinard/p/6677078.html>.
- [11] 大数据安全标准特别工作组. 人工智能安全标准化白皮书 (2019). 全国信息暗转标准化技术委员会.
- [12] 宋一凡. 自然语言处理的发展历史与现状 [j]. 中国高新科技.
- [13] 王威廉. 机器学习现在与未来. 卡内基梅陇大学计算机科学学院, 美国宾夕法尼亚州匹兹堡 15213.