文本分类技术基础知识篇（2019.03）

目录

1. 分类技术

1.1 基本算法

1.1.1 朴素贝叶斯算法

1.1.1.1 基本原理

朴素贝叶斯法利用贝叶斯定理首先求出联合概率分布，再求出条件概率分布。这里的朴素是指在计算似然估计时假定了条件独立。基本原理可以用下面的公式给出：

$$\mathrm{P(Y|X)} = \frac{P(Y)P(X|Y)}{P(X)}$$

$$\mathrm{P(X|Y)} = \mathrm{P}(X_1, X_2, \dots, X_n|Y) = P(X_1|Y)P(X_2|Y)\dots P(X_n|Y)$$

其中，P(Y|X)叫做后验概率，$P(Y)$叫做先验概率，$P(X|Y)$叫做似然概率，$P(X)$叫做证据。

1.1.1.2 多项式 NB

在文本相关的分类预测中，常见的朴素贝叶斯形式有多项式朴素贝叶斯和下面将要提到的贝努利朴素贝叶斯。多项式朴素贝叶斯的公式为：

- 训练阶段
  - 先验概率

$$\mathrm{P(C = c)} = \frac{\textit{属于类c的文档数}}{\textit{训练集文档总数}}$$

  - 条件概率

$$\mathrm{P}(w_i|\mathrm{c}) = \frac{\textit{词}w_i\textit{在属于类c的所有文档中出现次数}}{\textit{属于类c的所有文档中的词语总数}}$$

注：

（1） 条件概率$\mathrm{P}(w_i|\mathrm{c})$表示的是词$w_i$在类别 c 中的权重。

（2） 条件概率$\mathrm{P}(w_i|\mathrm{c})$的计算引入了位置独立性假设。也就是说在 c 类任意一篇文档内不同位置的词的条件概率是相等的。

（3） 上边两种概率值的计算都是利用的最大似然估计。它实际算出的是相对频率值，这些值能使训练数据的出现概率最大。

  - 拉普拉斯平滑（加 1 平滑）

$$\mathrm{P}(w_i|\mathrm{c}) = \frac{\textit{词}w\textit{在属于类c的所有文档中出现次数} + 1}{\textit{属于类c的所有文档中的词语总数} + \textbf{\textit{词汇表中词语总数}}}$$

加 1 平滑可以认为是采用均匀分布作为先验分布，即每个词项在每个类别中出现一次，然后根据训练数据对得到的结果进行更新。也就是说未登陆词的估计值为 1/词汇表长度。

- 预测阶段

$$\arg\max_{c\in C} P(c|w_1, w_2, \dots, w_n)$$
$$= \arg\max_{c\in C}[P(c)P(w_1, w_2, \dots, w_n|c)]$$
$$= \arg\max_{c\in C}[P(c)P(w_1|c)\,P(w_2|c)\dots P(w_n|c)]$$
$$= \arg\max_{c\in C}[logP(c) + logP(w_1|c) + \dots + logP(w_n|c)]$$

由计算式可以看出，预测阶段需要求和的项数为被预测文本所含词语个数+1

例：

	文档 ID	文档词列表	是否属于 China 类
训练集	1	Chinese Beijing Chinese	Yes
	2	Chinese Chinese Shanghai	Yes
	3	Chinese Macao	Yes
	4	Tokyo Japan Chinese	No
测试集	5	Chinese Chinese Chinese Tokyo Japan	?

训练：

先验概率 P(c=China)=3/4=0.75，P(c!=China)=1/4=0.25

类别 c=China 文档集词列表，词语总数 8

Chinese Beijing Chinese Chinese Chinese Shanghai Chinese Macao

类别 c!=China 文档集词列表，词语总数 3

Tokyo Japan Chinese

条件概率

词语序号	词语	$\mathrm{P}(w_i\|c = China)$	$\mathrm{P}(w_i\|c! = China)$
1	Chinese	(5+1)/(8+6)=0.429	(1+1)/(3+6)=0.222
2	Beijing	(1+1)/(8+6)=0.143	(0+1)/(3+6)=0.111
3	Shanghai	(1+1)/(8+6)=0.143	(0+1)/(3+6)=0.111
4	Macao	(1+1)/(8+6)=0.143	(0+1)/(3+6)=0.111
5	Tokyo	(0+1)/(8+6)=0.071	(1+1)/(3+6)=0.222
6	Japan	(0+1)/(8+6)=0.071	(1+1)/(3+6)=0.222

显然，训练阶段需要估计的参数个数为 词汇表长度*类别数量=6*2=12。

预测：
待预测文档词语列表
Chinese Chinese Chinese Tokyo Japn
P(c=China| Chinese Chinese Chinese Tokyo Japn)=
P(c=China)*P(Chinese|c=China)*P(Chinese|c=China)*P(Chinese|c=China)*P(Tokyo|c=China)*P(Japan|c=China)=0.75*0.429*0.429*0.429*0.071*0.071=0.000299
P(c!=China| Chinese Chinese Chinese Tokyo Japn)=
P(c!=China)*P(Chinese|c!=China)*P(Chinese|c!=China)*P(Chinese|c!=China)*P(Tokyo|c!=China)*P(Japan|c!=China)=0.25*0.222*0.222*0.222*0.222*0.222=0.000135
因为 P(c=China| Chinese Chinese Chinese Tokyo Japn)> P(c!=China| Chinese Chinese Chinese Tokyo Japn)模型认为待预测文档属于 c=China 类。

1.1.1.3 贝努利 NB

- 训练阶段
  - 先验概率

$$P(C=c)=\frac{\text{属于类}c\text{的文档数}}{\text{训练集文档总数}}$$

  - 条件参数

$$P(w_i|c)=\frac{\text{包含词}w_i\text{且属于类}c\text{的所有文档数}}{\text{属于类}c\text{的所有文档总数}}$$

  - 拉普拉斯平滑（加 1 平滑）

$$P(w_i|c)=\frac{\text{包含词}w_i\text{且属于类}c\text{的所有文档数}+1}{\text{属于类}c\text{的所有文档总数}+\textbf{类别总数}}$$

  未登录词的估计值为 1/类别总数。
- 预测阶段

$$\arg\max_{c\in C}P(c|w_1,w_2,\dots,w_n,\overline{w_{n+1}},\overline{w_{n+2}},\dots,\overline{w_{|V|}})$$
$$=\arg\max_{c\in C}\left[P(c)P(w_1,w_2,\dots,w_n,\overline{w_{n+1}},\overline{w_{n+2}},\dots,\overline{w_{|V|}}|c)\right]$$
$$=\arg\max_{c\in C}[P(c)P(w_1|c)\,P(w_2|c)\dots P(w_n|c)P(\overline{w_{n+1}}|c)P(\overline{w_{n+2}}|c)\dots P(\overline{w_{|V|}}|c)]$$
$$=\arg\max_{c\in C}[P(c)P(w_1|c)\,P(w_2|c)\dots P(w_n|c)(1-P(w_{n+1}|c))(1-P(w_{n+2}|c))\dots(1-P(w_{|V|}|c))]$$
$$=\arg\max_{c\in C}\left[logP(c)+logP(w_1|c)+\dots+logP(w_n|c)+\left(\mathbf{1}-\boldsymbol{logP}(\boldsymbol{w_{n+1}}|\boldsymbol{c})\right)+\left(\mathbf{1}-\boldsymbol{logP}(\boldsymbol{w_{n+1}}|\boldsymbol{c})\right)+\dots+\left(\mathbf{1}-\boldsymbol{logP}(\boldsymbol{w_{|V|}}|\boldsymbol{c})\right)\right]$$

  预测阶段需要的求和项数为 词汇表长度+1

例：

	文档 ID	文档词列表	是否属于 China 类
训练集	1	Chinese Beijing Chinese	Yes
	2	Chinese Chinese Shanghai	Yes
	3	Chinese Macao	Yes
	4	Tokyo Japan Chinese	No
测试集	5	Chinese Chinese Chinese Tokyo Japan	?

训练：
先验概率 P(c=China)=3/4=0.75，P(c!=China)=1/4=0.25
条件概率

词语序号	词语	c=China 包含w_i的文档数	c!=China 包含w_i的文档数	$P(w_i\|c=China)$	$P(w_i\|c!=China)$
1	Chinese	3	1	(3+1)/(3+2)=0.8	(1+1)/(1+2)=0.667
2	Beijing	1	0	(1+1)/(3+2)=0.4	(0+1)/(1+2)=0.333
3	Shanghai	1	0	(1+1)/(3+2)=0.4	(0+1)/(1+2)=0.333
4	Macao	1	0	(1+1)/(3+2)=0.4	(0+1)/(1+2)=0.333
5	Tokyo	0	1	(0+1)/(3+2)=0.2	(1+1)/(1+2)=0.667
6	Japan	0	1	(0+1)/(3+2)=0.2	(1+1)/(1+2)=0.667

显然，训练阶段需要估计的参数个数为 词汇表长度*类别数量=6*2=12。
预测：
待预测文档词语集合
Chinese Tokyo Japn
P(c=China| Chinese Tokyo Japan)=
P(c=China)*P(Chinese|c=China)*P(Tokyo|c=China)*P(Japan|c=China)*(1-P(Beijing|c=China))+(1-P(Shanghai|c=China))*(1-P(Macao|c=China))=0.75*0.8*0.2*0.2*(1-0.4)*(1-0.4)*(1-0.4)=0.005
P(c!=China|Chinese Tokyo Japan)=
P(c!=China)*P(Chinese|c!=China)*P(Tokyo|c!=China)*P(Japan|c!=China)*(1-P(Beijing|c!=China))*(1-P(Shanghai|c!=China))*(1-P(Macao|c!=China))=0.25*0.667*0.667*0.667*(1-0.333)*(1-0.333)*(1-0.333)=0.022
模型认为待预测文档属于 c!=China 类
注：
（1） 当对长文档进行预测时，采用贝努利模型往往会因为某个词比如 China 在文档中出现一次而将整篇文档归于 China 类。
（2） 词典中没有在被预测文档中出现的的词，对于多项式模型的预测没有影响，但是对于贝努利模型有影响，因为它的预测公式中考虑了这样的词发生的可能性。

1.1.1.4 模型比较

	多项式模型	贝努利模型
事件模型	词条生成模型	文档生成模型
随机变量	X = t，当且仅当 t 出现在给定位置	$U_t = 1$，当且仅当 t 出现在文档中
文档表示	d =< $t_1,,,t_k,\dots,t_{nd}>, t_k \in V$	d =< $e_1,,,e_i,\dots,e_M>, e_i \in \{0,1\}$
参数估计	$\hat{P}(X = t\|c)$	$\hat{P}(U_i = e\|c)$
决策规则：最大化	$\hat{P}(c)\prod_{1\le k\le n_d}\hat{P}(X = t_k\|c)$	$\hat{P}(c)\prod_{t_i\in V}\hat{P}(U_i = e_i\|c)$
词项多次出现	考虑	不考虑
文档长度	能处理更长文档	最好处理短文档
特征数目	能够处理更多特征	特征数目较少效果更好
训练需要估计的参数个数	|V|*类别数量	|V|*类别数量
预测需要求和的项数	被预测文档所含词语数+1	词汇表长度+1

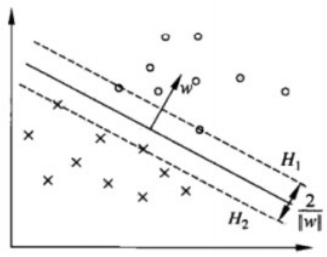
NB 算法适用于下边描述的场景（1）数量大（2）对概念漂移的鲁棒性有一定要求（3）不值得为提高一点准确率而引入更繁琐策略。

1.1.2 Linear SVC

基于支持向量机的分类方法主要用于解决**二元模式**分类问题。SVM 的基本思想是在向量空间中找到一个决策平面，这个平面能“最好”地分割两个类别中的数据点。支持向量机分类法就是要在训练集中找到具有**最大类间隔**的决策平面。这也是它与感知机的区别。

由于支持向量机算法是基于两类模式识别问题的，因而，对于多类模式识别问题通常需要建立多个两类分类器。与先行判别函数一样，它的结果强烈地依赖于已知模式样本集的构造，当样本容量不大时，这种依赖性尤其明显。此外，将分界面定在最大类间隔的中间，对于许多情况来说也不是最优的。对于线性不可分问题也可以采用类似于广义线性判别函数的方法，通过事先选择好的非线性映射将输入模式向量映射到一个高维空间，然后在这个高维空间中构造最优分类超平面。

1.1.2.1 术语



间隔边界：图中虚线H_1，H_2
支持向量：间隔边界上的实例点
分离超平面：两条间隔边界的对称轴
间隔：两个间隔边界之间的距离
线性可分：存在超平面可以将两类样本分开（用数学式子描述就是支持向量机凸优化问题的约束条件）
上边的术语在平面上的数学描述为：
H_1的直线方程为：$\mathrm{wx+b=1}$，令$\mathrm{w=(A,B)}$，$C_1 = b - 1$，$\mathrm{x=(x,y)}$，则直线方程为$\mathrm{Ax+By}+C_1 = 0$
H_2的直线方程为：$\mathrm{wx+b=-1}$，令$\mathrm{w=(A,B)}$，$C_2 = b + 1$，$\mathrm{x=(x,y)}$，则直线方程为$\mathrm{Ax+By}+C_2 = 0$
超平面（间隔边界）的直线方程为：$\mathrm{wx+b=0}$
间隔（距离）的计算：$\mathrm{d}=\frac{|C_1-C_2|}{\sqrt{A^2+B^2}}=\frac{2}{\|w\|}$
注：上边的直线方程中的 y，都是实例的第 2 个维度坐标，并非类别标记。

1.1.2.2 核心思想

核心思想是寻找一个超平面，使得两类实例点位于该超平面的两侧。由于这样的超平面不唯一，考虑到模型的鲁棒性，认为超平面应当使得所有实例点到超平面的距离最大。

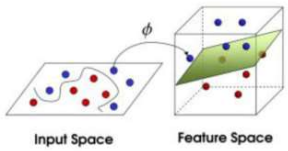
如果超平面$\mathrm{wx+b=0}$是间隔边界，则所有实例$(\mathbf{x}_i, y_i)$应该满足不等式$y_i(\mathrm{w}\mathbf{x}+\mathrm{b})-1\ge 0$，此不等式就是约束条件。距离最大就是目标函数$\max_{w,b}\frac{2}{\|w\|}$，它等价于$\min_{w,b}\frac{\|w\|^2}{2}$。

1.1.2.3 线性可分支持向量机与线性支持向量机

	线性可分支持向量机	线性支持向量机
原始问题	$\min\limits_{w,b}\frac{\|w\|^2}{2}$ s.t. $y_i(wx_i+b)\geq 1, i=1,2,\dots,N$	$\min\limits_{w,b}\frac{\|w\|^2}{2}+C\sum\limits_{i=1}^{N}\xi_i$ s.t. $y_i(wx_i+b)\geq 1-\xi_i, i=1,2,\dots,N$ $\xi_i\geq 0, i=1,2,\dots,N$ 训练过程就是在最小化ξ_i，因为ξ_i表示的是对分隔边界移动的距离。之所以移动分隔边界就是使误分类点尽量少。 C 值叫做惩罚系数，C 值小，对误分点容忍度高，但损失精度；C 值大，对误分点容忍度低，但容易过拟合。
对偶问题	$\min\limits_{\alpha}\frac{1}{2}\sum\limits_{i=1}^{N}\sum\limits_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)+\sum\limits_{i=1}^{N}\alpha_i$ s.t. $\sum_{i=1}^{N}\alpha_i y_i=0, \alpha_i\geq 0$	$\min\limits_{\alpha}\frac{1}{2}\sum\limits_{i=1}^{N}\sum\limits_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)+\sum\limits_{i=1}^{N}\alpha_i$ s.t. $\sum_{i=1}^{N}\alpha_i y_i=0, \alpha_i\geq 0$ $0\leq\alpha_i\leq C, i=1,2,\dots,N$
对偶问题解表示原始问题解	$w^*=\sum\limits_{i=1}^{N}\alpha_i^* y_i x_i$ $b^*=y_j-\sum_{i=1}^{N}\alpha_i^* y_i(x_i\cdot x_j)$，其中 j 是满足$\alpha_j^*>0$的下标	$w^*=\sum\limits_{i=1}^{N}\alpha_i^* y_i x_i$ $b^*=y_j-\sum_{i=1}^{N}\alpha_i^* y_i(x_i\cdot x_j)$，其中 j 是满足$0<\alpha_j^*<C$的下标

1.1.2.4 非线性支持向量机

对于文本分类来说，一般情况下实例点都是线性可分的，也就是说即使采用非线性分类器，也不会使分类精度有明显提升。因此这里只是简单说明非线性支持向量机的原理。我们用一张非线性变换图来形象说明核函数的作用。



从图中可以看到原来的实例点在一个 2 维空间上，显然在这里只能使用一条曲线将两类实例点分离到曲线两侧。那么怎么才能用一个线性超平面将两类实例点分开呢？答案是使用非线性函数ϕ，将 2 维空间中的实例点映射到 3 维空间，也就是图中的 feature space，这时就可以在 3 维空间内找到一个平面，将两种实例分离到平面的两侧。需要说明的是，当把数据映射到高维空间后，每一个实例点都需要更多的数值来表示，这样在计算对偶公式中的内积时会使计算量增大，这是不能忍受的，怎样解决这个问题呢？根据 mercer 定理，针对某一种ϕ可以找到一个核函数 K，用这个核函数 K 在低维空间中计算出某两个实例的值，这个值刚好就是这两个实例在高维空间中的内积值，这样就可以避免在高维空间计算内积，从而减少了计算量。

2.2.2.5 LR 与 LinearSVC

模型名称	预测函数	成本函数	迭代公式
线性回归	$h_\theta(x)=\theta^T x$	$\frac{1}{2m}\sum\limits_{i=1}^{m}\left(h(x^{(i)})-y^{(i)}\right)^2$	$\theta_j=\theta_j-\frac{\alpha}{m}\sum\limits_{i=1}^{m}\left(h(x^{(i)})-y^{(i)}\right)x_j^{(i)}$
逻辑回归	$h_\theta(x)=\frac{1}{1+e^{-\theta^T x}}$	$-\frac{1}{m}\left[\sum\limits_{i=1}^{m}y^{(i)}\log\left(h(x^{(i)})\right)+\left(1-y^{(i)}\right)\log\left(1-h(x^{(i)})\right)\right]$	$\theta_j=\theta_j-\frac{\alpha}{m}\sum\limits_{i=1}^{m}\left(h(x^{(i)})-y^{(i)}\right)x_j^{(i)}$
Linear SVC	$h_\theta(x)=sign(\theta^T x)$	$\sum\limits_{i=1}^{m}\left[1-y^{(i)}\theta^T x^{(i)}\right]_+$	

那么 LR 与 SVC 有什么区别呢？从损失函数讲 SVC 损失函数是$[1-z]_+$，LR 损失函数是$\log(1+e^{-z})$。两个损失函数的目的都是增加对分类影响较大的实例点的权重，减少与分类关系较小的实例点的权重。区别是 SVC 关注的是离决策面较近的实例点，LR 关注的是所有实例点，它通过非线性映射降低了离决策面较远点的权重，从而相对提升了离决策面较近点的权重。

逻辑回归的详细说明，二项逻辑斯谛回归模型，用于解决二类分类，它将线性函数 wx 转换为概率。

$$P(Y=1|x)=\frac{e^{wx}}{1+e^{wx}}$$

$$P(Y=0|x)=\frac{1}{1+e^{wx}}$$

逻辑回归模型的参数求解等价于似然函数的最优化问题，求解方法有改进迭代尺度法，梯度下降法，拟牛顿法。

多项逻辑斯谛回归模型用于多类分类

$$P(Y=k|x)=\frac{e^{w_k x}}{1+\sum_{k=1}^{K-1}e^{w_k x}},\quad k=1,2,\dots,K-1$$

1.1.3 SGD 线性分类算法

梯度下降法按照计算训练误差时使用训练数据的方式分为以下三种：批量梯度下降法（Batch gradient descent, BGD），随机梯度下降法（Stochastic gradient descent, SGD），小批量梯度下降法（Mini-Batch gradient descent, MBGD）。

1. 批量梯度下降法。每次迭代使用所有的样本，这样做的好处是每次迭代都顾及了全部的样本，考虑的是全局最优化。需要注意的是这个名字并不确切，但是机器学习领域中都这样称。它的缺点是每次迭代都要计算训练集中所有样本的训练误差，当数据量很大时，这种方法效率不高。

2. 随机梯度下降法。每次迭代都随机从训练集中抽取出 1 个样本，在样本量极其大的情况下，可能不用抽取出所有样本，就可以获得一个损失值在可接受范围之内的模型了。**缺点是由于单个样本可能会带来噪声，导致并不是每次迭代都向着整体最优方向前进。**
3. 小批量梯度下降法。它介于批量梯度下降法与随机梯度下降法之间。每次迭代随机从训练集抽取一定数量的数据进行训练。

1.2 多类目预测

根据类别之间是否互斥，可以将多类问题的分类细分为两类问题。第 1 类问题是多标签分类问题，它指的是一个文本可以同时属于多个类别，类别之间不是互斥的。第 2 类问题是单标签问题，它指的是一个文本最多只能属于 1 个类别，即多个类别之间是互斥的。

1.2.1 单标签分类步骤

处理单标签分类可以直接使用多分类模型，比如 k 近邻、朴素贝叶斯、决策树等。对于二类分类模型需要使用组合策略，算法描述如下：
（1）对每个类别建立一个分类器，此时训练集包含所有属于该类的文档和所有不属于该类的文档
（2）给定测试文档，分别使用每个分类器进行分类
（3）将文档分配给得分最高的类、置信度最高的类或概率最大的类
建立分类器的方法可以有两种，分别为 one-vs-one 和 one-vs-rest。**one-vs-one** 方法就是在建立分类器时随机抽取两个类别（不考虑顺序），因此分类器的数量为 n(n-1)/2（n 为类别数量）。**one-vs-rest** 方法就是在建立分类器时选定一个类别作为一组，其余类别作为一组来构建分类器，因此分类器的数量为 n。

1.2.2 多标签分类步骤

（1） 对每个类别建立一个分类器，此时训练集包含所有属于该类的文档和所有不属于该类的文档
（2） 给定测试文档，分别使用每个分类器进行分类，每个分类器的分类结果并不影响其他分类器的结果

1.3 模型选择

1.3.1 模型选择之交叉验证法

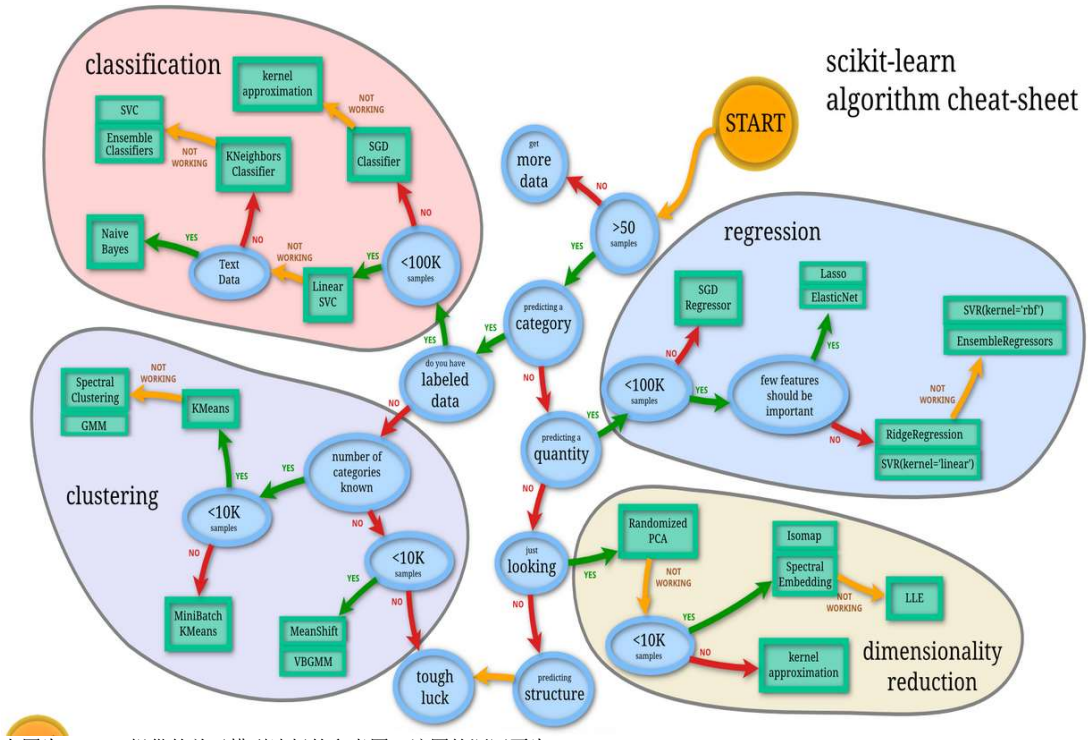
首先回答下边这个问题。为什么要将数据分为训练集、验证集（交叉验证数据集）和测试集？因为三种数据集有各自的用途。训练集用来训练模型的参数，验证集用来选择最合适的模型，测试集用来测试模型的准确性。一般情况下，我们将数据按 6:2:2 分成训练集、交叉验证集和测试集。

通过回答上边的问题我们知道，模型的选择是通过比较不同模型在验证集上的准确性来实现。也就是说在验证集上准确性最高的模型就认为是最优的模型。在验证集上评估模型准确性的方法有以下三种：简单交叉验证法，k 折交叉验证法，留一法。为了方便说明总结成下表形式：

类型	处理过程	优缺点
简单交叉验证	数据被随机分为两组，分别作为训练集和验证集，在训练集上训练，在验证集上计算准确率。	随机将数据划分为训练集和验证集，那么在验证集上的准确率会随着划分比例的变化而变化，因此这样得到的验证集上的准确率通常不具有说服力。
K 折交叉验证	数据被均等分为 k 组，每次取 1 组作为验证集，其他组作为训练集。分别在 k 个训练集上训练，并对应在 k 个验证集上计算出 k 个准确率，最后求平均值作为模型在验证集上最终的准确率。	有效避免了过拟合和欠拟合，最终结果更具说服力。
留一交叉验证	每次取 1 个实例作为测试集，其他实例作为训练集。分别在 n（实例总数）个训练集上训练，并对应在 n 个验证集上计算出 n 个准确率，最后求平均值作为模型在验证集上最终的准确率。	结果相对可靠。确保实验过程可复现。缺点计算成本高，当数据量很大时，该方法实用性不强。

1.3.2 模型选择之经验法

1.3.2.1 模型选择图



上图为 sklearn 提供的关于模型选择的参考图。该图的源网页为（https://scikit-learn.org/stable/tutorial/machine_learning_map/）。图中左上为分类算法，左下为聚类算法。在聚类算法中，GMM、VBGMM 为语音处理算法。MeanShift 为图像处理算法。因此应用于文本的聚类算法有 KMeans（数据量小于 1 万）、MiniBatchKMeans（数据量大于 1 万）、Spectral Clustering（数据量小于 1 万），它们都是用来解决已知簇数量的算法，对于未知簇数量的情况，可以采用增量聚类算法比如 single-pass。在分类算法中当数据量小于 10 万时，使用分类器 LinearSVC 或者 NaiveBayes。当数据量大于 10 万时采用随机梯度下降线性分类器。

1.3.2.2 模型选择表

模型	优点	缺点	应用
NB	1.**大数据量** 2.适用多类问题分类 3.可以进行增量训练 4.结果易解释便于分析误判	1.**特征之间有关联时效果受影响**。比如两词短语所表达的语义与单独的词语表达的语义不同时。或者两词短语不同的排列顺序具有不同的语义时。	1.欺诈检测 2.垃圾邮件 3.文本分类 4.情感分类
LR	1.预测速度快 2.模型占用内存小 3.多种正则化方法避免过拟合 4.不必担心特征间存在相关性	1.**不适合高维数据** 2.容易欠拟合 3.适用于二类分类 4.数据必须线性可分	1.检索排序 2.信用评估 3.产品收益 4.地震预测
KNN	1.非线性分类 2.训练时间快 3.异常数据不敏感 4.可以进行增量训练	1.**不平衡数据效果差** 2.模型占用内存大 3.**计算量大（相似度距离）** 4.**k 值选择没有理论指导** 5.容易过拟合	1.文本分类 2.聚类分析
决策树	1.易于可视化 2.易于制定规则 3.同时处理标称型和离散型数据	1.容易过拟合 2.属性选择时易受取值数量影响	1.企业管理 2.投资
SVM	1.高维数据 2.**小样本** 3.非线性 4.**泛化能力强** 5.模型占用内存小	1.**二分类** 2.核函数不好找（当然一般在文本分类中使用线性 SVM）	1.文本分类

1.3.3 实践中的经验

1. 一个普遍的事实就是，采用领域相关的文本特征在效果上会比采用新的机器学习方法获得更大的提升。
2. Jackson 和 Moulinier（2002）指出：“对数据的理解是分类成功的关键之一”。
3. 当面对一个具体的分类需求时，第一个要问的问题就是：训练数据（已标注的数据）有多少？如果有足够多的时间用于系统实现的话，那么大部分时间可能要花在数据的准备上。
4. 在没有标注数据的情况下，一般首先采用编制规则的方法。一个基本合理的估计数字是每个类别需要标注两天的时间。
5. 在已标注数据较少的情况下，一般采用高偏差分类器，比如贝叶斯。当然，无论采用何种模型，模型的质量始终会因训练数据有限而受到不利影响。
6. 快速标注数据的方法 bootstrapping 方法。将分类器不太好分的文本交给人工进行标注。
7. 快速进行错误纠正的方法是在分类模型之上，再建立一个基于布尔规则的分类器。
8. 如果具有极大规模的数据，那么分类器的选择也许对最后的结果没有什么影响。一般可以从训练规模扩展性或运行效率上来选择分类器。
9. 一个通用的经验法则是，训练数据规模每增加一倍，那么分类器的效果将得到线性的提高。
10. 对于类别数目很多分类问题，可以采用分层策略。
11. 将特殊字符串（比如 ISBN 号，化学式等）按照类别统一成一种符号。
12. 用短语作为特征。用命名实体作为特征。

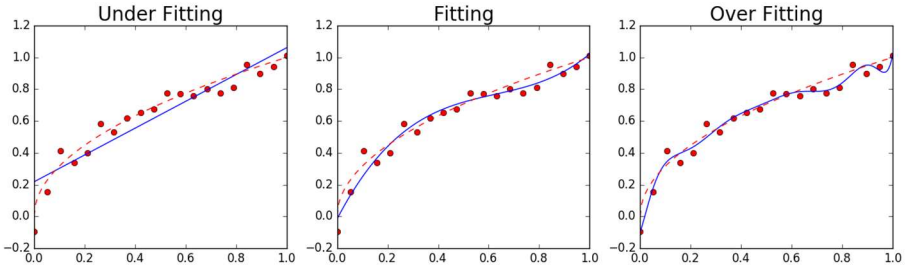
1.4 模型评价

1.4.1 训练误差与测试误差

训练误差就是模型在训练集上的误差平均值，度量了模型对训练集拟合的情况。训练误差大说明对训练集特性学习得不够，训练误差太小说明过度学习了训练集特性，容易发生过拟合。**测试误差**是模型在测试集上的误差平均值，度量了模型的泛化能力。在实践中，希望测试误差越小越好。

1.4.2 过拟合与欠拟合

如同上一小结的阐述，**过拟合**就是模型过度学习了训练集所有特性，导致模型认为训练集中的某些特性也是潜在测试实例具有的一般性质。从误差角度来说，过拟合时训练误差小但测试误差却很大。**欠拟合**就是说模型尚未学习完整训练集实例的普适特性。从误差角度来说，欠拟合时训练误差大，测试误差也大。为了防止过拟合出现，需要根据不同模型采用不同的方法。对于优化损失函数的模型比如感知机、逻辑回归、SVM 等可以在损失函数中加入正则化项（罚项），正则化项一般是模型参数的范数。对于决策树这样的模型，可以通过剪枝达到避免过拟合的目的。下面用下边的实例来说明欠拟合和过拟合：



上图中，左边是欠拟合（underfitting），也称为高偏差（high bias）因为我们试图用一条直线来拟合样本数据。右边是过拟合（overfitting），也称为高方差（high variance），用了十阶多项式来拟合数据，虽然模型对现有的数据集拟合得很好，但对新数据预测误差却很大。只有中间的模型较好地拟合了数据集，可以看到虚线和实线基本重合。示例代码如下：

```
#coding:utf-8
"""
过拟合和欠拟合演示
"""

import matplotlib.pyplot as plt
import numpy as np


# 画出拟合出来的多项式所表达的曲线以及原始的点
def plot_polynomial_fit(x, y, order):
    p = np.poly1d(np.polyfit(x, y, order))
    t = np.linspace(0, 1, 200)
    plt.plot(x, y, 'ro', t, p(t), '-', t, np.sqrt(t), 'r--')
    return p


# 生成 20 个点的训练样本
n_dots = 20
x = np.linspace(0, 1, n_dots)  # [0, 1] 之间创建 20 个点
y = np.sqrt(x) + 0.2*np.random.rand(n_dots) - 0.1;

plt.figure(figsize=(18, 4))
titles = ['Under Fitting', 'Fitting', 'Over Fitting']
models = [None, None, None]
```

```
for index, order in enumerate([1, 3, 10]):
    plt.subplot(1, 3, index + 1)
    models[index] = plot_polynomial_fit(x, y, order)
    plt.title(titles[index], fontsize=20)
plt.show()
```

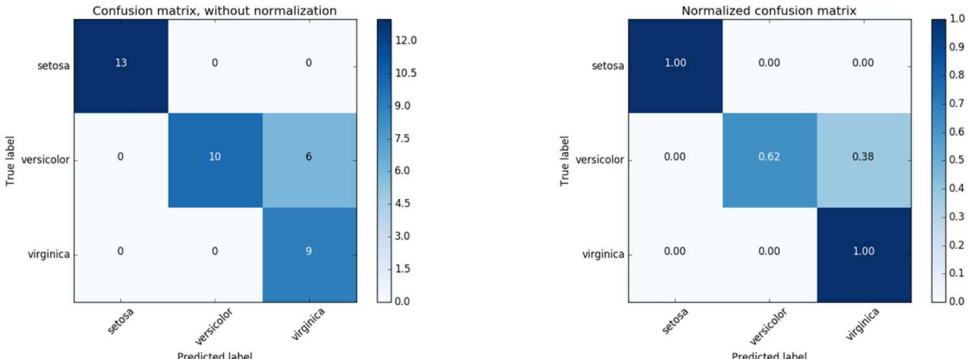
1.4.3 混淆矩阵

混淆矩阵对角线上的元素表示的是正确预测各类别的实例数量，而非对角线上的元素表示误分类实例数量。对角线元素值之和越大，表示正确预测实例数量越多。

预测类别 / 实际类别	money-fx	trade	interest	wheat	corn	grain
money-fx	95	0	10	0	0	0
trade	1	1	90	0	1	0
interest	13	0	0	0	0	0
wheat	0	0	1	34	3	7
corn	1	0	2	13	26	5
grain	0	0	2	14	5	10

上表中有 14 篇属于 grain 类的文档被误分到 wheat 类中。

混淆矩阵的 sklearn 示例。这里在 iris 数据集上训练一个分类器，并用混淆矩阵来评价。图像分别对未经规范化和经过规范化的混淆矩阵进行了可视化。这种规范化后的可视化有利于评价不平衡分类。这里的分类器没有达到其应有的准确率。因为正则化参数 C 没有选择好。在真实的应用中，通常通过`grid_search`来确定 C 的值。




```
"""
================
Confusion matrix
================

混淆矩阵用法的例子。这里在 iris 数据集上训练一个分类器，并用混淆矩阵来评价。
矩阵对角线上的元素表示的是正确预测各类别的实例数量，而非对角线上的元素表示误分类实例数量。
对角线元素值之和越大，表示正确预测实例数量越多。

图像分别对未经规范化和经过规范化的混淆矩阵进行了可视化。
这种规范化后的可视化有利于评价不平衡分类。

这里的分类器没有达到其应有的准确率。因为正则化参数 C 没有选择好。
在真实的应用中，通常通过`grid_search`来确定 C 的值。

"""

print(__doc__)

import itertools
import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
class_names = iris.target_names

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
# Run classifier, using a model that is too regularized (C too low) to see
# the impact on the results
classifier = svm.SVC(kernel='linear', C=0.01)
y_pred = classifier.fit(X_train, y_train).predict(X_test)

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix, without normalization')

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

1.4.4 精确率、准确率、召回率、F 值、宏平均和微平均

二类分类的混淆矩阵

真阳性 tp	假阴性 fn
假阳性 fp	真阴性 tn

1. 准确率

$$\text{accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

考虑各个类别，计算整体的平均准确性。对于不平衡分类，准确率并不是一个好的衡量指标。这是因为一个将所有文档都不归于小类的分类器会获得很高的准确率，但是这显然并不能说明系统实际的准确性。因此对于不平衡分类来说，精确率、召回率和 F 值才是更好的衡量指标。该指标可以通过 sklearn 分类模型实例的 score 方法得到。

2. 精确率

$$\text{precision} = \frac{tp}{tp + fp}$$

度量分类器在某一个类别上的预测精确性。

3. 召回率

$$\text{recall} = \frac{tp}{tp + fn}$$

度量分类器在某一个类别上的预测覆盖面。

4. F 值

$$\text{F} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

度量分类器在某一个类别上的预测精确性以及覆盖面。它是精确率和召回率的调和平均值。

5. 宏平均与微平均

当评价多类目分类器的效果时，还经常采用宏平均和微平均两个度量方法。下表以两个类别为例说明计算方法。

类别 1			类别 2			缓冲表		
	实际 yes	实际 no		实际 yes	实际 no		实际 yes	实际 no
判定 yes	10	10	判定 yes	90	10	判定 yes	100	20
判定 no	10	970	判定 no	10	890	判定 no	20	1860

1) 宏平均

$$\text{macro avg} = \frac{\left(\frac{10}{10+10} + \frac{90}{90+10}\right)}{2} = 0.7$$

2) 微平均

$$\text{micro avg} = \frac{100}{100+20} = 0.83$$

微平均实际上是文档集中大类（含有很多数据的类目）上的一个效果度量指标，如果要度量小类上的效果，往往需要计算宏平均指标。

sklearn 中通过 classification_report 实现上边后 5 种指标，示例如下：

```
>>> y_true = [-1, 0, 1, 1, -1]
>>> y_pred = [-1, -1, 1, 0, -1]
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [1, 0, 0],
       [0, 1, 1]])
>>> from sklearn.metrics import classification_report
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
              precision    recall  f1-score   support

     class 0       0.67      1.00      0.80         2
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.50      0.67         2

   micro avg       0.60      0.60      0.60         5
   macro avg       0.56      0.50      0.49         5
weighted avg       0.67      0.60      0.59         5
```

这里混淆矩阵中行以及列的序号对应的类别整数依次是预测值去重升序排列后对应的类别整数值。因此第 0 行对应的类别是-1，第 1 行对应的类别是 0，第 2 行对应的类别是 1，列与此相同。classification_report 中 target_names 列表中的类别名称依次对应的是预测值去重生序排列后对应的类别整数值。因此’class 0’对应的类别是-1，’class 1’对应的类别是 0，’class 2’对应的类别是 1。

1.4.5 ROC 曲线与 AUC 值

ROC 曲线和 AUC 值是处理不平衡分类问题的评价方法。显示分类器真正率和假正率之间折中的一种图形化方法。真正率和假正率的定义将在下边介绍。一个好的分类模型的 ROC 曲线应尽可能靠近面积为 1 的正方形的左上角。AUC 值是 ROC 曲线下的面积。AUC 值越大，分类器效果越好。

首先给出二类分类的混淆矩阵

真阳性 TP	假阴性 FN
假阳性 FP	真阴性 TN

真正率（true positive rate, TPR）或灵敏度（sensitivity）定义为被模型正确预测的正样本的比例 TPR=TP/(TP+FN)。

假正率（false positive rate, FPR）定义为被预测为正类的负样本比例 FPR=FP/(TN+FP)。

在度量不平衡分类的分类模型时，将稀有类定义为正类，常见类定义为负类。

ROC 曲线的绘制过程：

为了能够绘制 ROC 曲线，分类器需要能提供预测类别的得分值，用来对预测为正类的实例按得分排序，最不肯定的排在前，最肯定的排在后。需要注意的是，这个得分是预测为正类（稀有类）的分值，而不是正、负类中得分最高的值。

（1） 让模型对每一个实例进行预测，记录正类得分，并按得分将实例升序排列。

（2） 从排序列表中按顺序选择第 1 个得分最小的记录，从该记录开始到列表结束的所有记录都被指定为正类，其他实例指定为负类，计算混淆矩阵并计算 TPR，FPR。此时，TPR=FPR=1。

（3） 从排序列表中选择下 1 个记录，从该记录开始到列表结束的所有记录都被指定为正类，其他实例指定为负类，计算混淆矩阵并计算 TPR，FPR

（4） 重复步骤（3），直到列表中所有实例都被选择过。

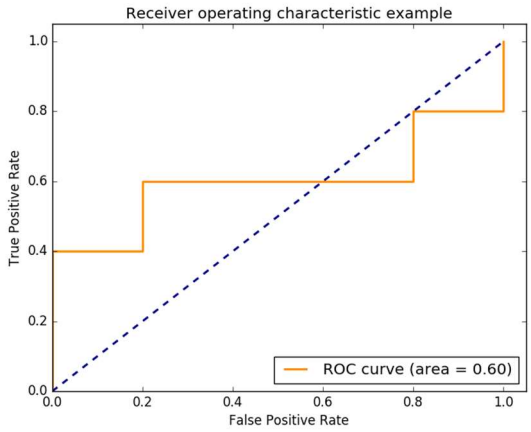
（5） 以 FPR 为横轴，TPR 为纵轴，描点绘制 ROC 曲线。

示例

下表中，每一列表示一个实例，已经按照预测为正类的得分升序排列。第 1 行为实例的实际类别，第 2 行为实例被模型预测为正类的得分。请计算出第 2 行之后的各行表示的混淆矩阵元素值以及 TPR、FPR 值。

实例的实际类别	+	-	+	-	-	-	+	-	+	+
模型预测为正类得分	0.25	0.43	0.53	0.76	0.85	0.85	0.85	0.87	0.93	0.95
TP	5	4	4	3	3	3	3	2	2	1
FP	5	5	4	4	3	2	1	1	0	0
TN	0	0	1	1	2	3	4	4	5	5
FN	0	1	1	2	2	2	2	3	3	4
FPR	1	0.8	0.8	0.6	0.6	0.6	0.6	0.4	0.4	0.2
TPR	1	1	0.8	0.8	0.6	0.4	0.2	0.2	0	0

解：首先选择第 1 个实例，按照绘制过程的第（2）个步骤，此时所有实例都被指定为+，则比较表中第 1 行的实际类别，可以计算出 TP=5, FP=5, TN=0, FN=0, TPR=1, FPR=1。将计算得到的值填入表中第 1 列相应位置。接着按照第（3）个步骤，选择第 2 个实例，此时从第 2 到第 10 的 8 个实例指定为正类，其余实例即第 1 个实例指定为负类，计算 TP=4,FP=5,TN=0,FN=1,TPR=4/(4+1)=0.8, FPR=5/(5+0)=1。依次类推计算第 3-8 列的各行元素值。接下来便可以 FPR 为横轴，TPR 为纵轴，描出表中给(fpr,tpr)点，绘制模型的 ROC 曲线。




```
#coding:utf-8
"""
绘制 ROC 曲线，计算 AUC 值示例
"""

import numpy as np
from sklearn.metrics import roc_curve,auc
import matplotlib.pyplot as plt

# 实际类别只能取{0,1}或{1,-1}
y = np.array([1,0,1,0,0,0,1,0,1,1])
# 对应预测为正（即 1）的得分。注意：得分相同的实例只保留一个
scores = np.array([0.25,0.43,0.53,0.76,0.85,0.86,0.87,0.89,0.93,0.95])
# pos_label 假定为正类的类别标记，这里是 1
fpr, tpr, thresholds = roc_curve(y_true = y, y_score = scores, pos_label=1)
print("tpr=", tpr)
print("fpr=", fpr)
print("thresholds=", thresholds)
# 计算 auc 值
roc_auc = auc(fpr, tpr)

plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',lw=lw, label='ROC curve (area = %.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.05])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

```
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

问题：在机器学习中 AUC 和 accuracy 有什么内在关系？
auc 代表的是分类或者排序能力，与分类阈值无关；准确率是和阈值有关的。auc 高，准确率低，可能的原因是分类阈值的选择引起的。极端来讲，默认阈值是 0.5，但模型输出的值全部小于 0.5，那正样本是全错的；但由于 auc 很高，正负样本还是可以分开，把分类阈值调小即可。auc 低，准确率高，这种一般发生在分布不平衡的问题中。比例少的那类分类错误多，但由于数量少整体的准确率还是很高，但代表分类能力的 auc 就会很低。参考：https://www.zhihu.com/question/313042288

1.5 简单示例

1.5.1 英文新闻文本分类

```
#coding:utf-8
"""
朴素贝叶斯示例-英文新闻文本分类
"""
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB,BernoulliNB
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.feature_extraction.text import TfidfVectorizer

## 导入数据集以及数据集的简单处理
# 导入全部的训练集和测试集
news = fetch_20newsgroups(subset="all")
# 打印类目名称列表
print("类目名称列表\n",u'\n'.join(news.target_names))
# 打印类目数量
print("类目数量\n", len(news.target_names))
# 打印数据 X 量
print("训练集文本数量\n", len(news.data))
# 打印类标 Y 量
print("标记了类别的文本数量\n", len(news.target))
# 打印第 0 篇文本
print("第 1 篇文本内容\n", news.data[0])
# 打印类目序号
print("第 1 篇文本的类别序号\n", news.target[0])
# 打印类目序号所对应的类目名称
print("第 1 篇文本的类别名称\n", news.target_names[news.target[0]])
# 数据集切分
x_train, x_test, y_train, y_test = train_test_split(news.data, news.target)
# 向量化
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(x_train)
X_test = vectorizer.transform(x_test)
print("="*50)
# 构建多项式朴素贝叶斯实例
mul_nb = MultinomialNB()
# 训练模型
mul_nb.fit(X_train, y_train)
# 打印测试集上的分类报告
print("分类报告\n", classification_report(mul_nb.predict(X_test), y_test))
# 打印测试集上的混淆矩阵
print("混淆矩阵\n", confusion_matrix(mul_nb.predict(X_test), y_test))
```

1.5.2 英文影评情感分类

```
#coding:utf-8
"""
朴素贝叶斯影评情感分析
"""
from nltk.corpus import movie_reviews
from sklearn.model_selection import StratifiedShuffleSplit
import nltk
from nltk.corpus import stopwords
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures

def get_data():
    """
    获取影评数据
    """
    dataset = []
    y_labels = []
    # 遍历类别
    for cat in movie_reviews.categories():
        # 遍历每个类目的评论 id
        for fileid in movie_reviews.fileids(cat):
            # 读取评论词语列表
            words = list(movie_reviews.words(fileid))
            dataset.append((words, cat))
            y_labels.append(cat)
    return dataset, y_labels

def get_train_test(input_dataset, ylabels):
    """
    划分数据为训练集和测试集
    """
    train_size = 0.7
    test_size = 1-train_size
    stratified_split = StratifiedShuffleSplit(n_splits=10, test_size=test_size, random_state=77)
    for train_indx, test_indx in stratified_split.split(input_dataset, ylabels):
        train = [input_dataset[i] for i in train_indx]
        train_y = [ylabels[i] for i in train_indx]

        test = [input_dataset[i] for i in test_indx]
        test_y = [ylabels[i] for i in test_indx]

    return train, test, train_y, test_y

def build_word_features(instance):
    """
    构建特征词典
    one-hot，特征名称为词语本身，特征值为 bool 类型值
    """
    # 存储特征的词典
    feature_set = {}
    # instance 的第 1 个元素为词语列表
    words = instance[0]
    # 填充特征词典
    for word in words:
        feature_set[word] = 1
    # instance 的第 2 个元素为类别名称
    return feature_set, instance[1]

def build_negate_features(instance):
    """
    如果一个词语前有否定关键词(not 或 no)修饰，则对词语加前缀 Not_，否定关键词不再被添加到特征词典
    """
    # Retreive words, first item in instance tuple
    words = instance[0]
    final_words = []
    # A boolean variable to track if the
    # previous word is a negation word
    negate = False
    # List of negation words
    negate_words = ['no', 'not']
    # On looping throught the words, on encountering
    # a negation word, variable negate is set to True
    # negation word is not added to feature dictionary
    # if negate variable is set to true
    # 'Not_' prefix is added to the word
```

```
    for word in words:
        if negate:
            word = 'Not_' + word
            negate = False
        if word not in negate_words:
            final_words.append(word)
        else:
            negate = True
    # Feature dictionary
    feature_set = {}
    for word in final_words:
        feature_set[word] = 1
    return feature_set, instance[1]

def remove_stop_words(in_data):
    """
    去除停用词
    Utility function to remove stop words
    from the given list of words
    """
    stopword_list = stopwords.words('english')
    negate_words = ['no', 'not']
    # We dont want to remove the negate words
    # Hence we create a new stop word list excluding
    # the negate words
    new_stopwords = [word for word in stopword_list if word not in negate_words]
    label = in_data[1]
    # Remove stopw words
    words = [word for word in in_data[0] if word not in new_stopwords]
    return words, label

def build_keyphrase_features(instance):
    """
    构建短语特征
    """
    feature_set = {}
    # 应用 map 迭代器
    instance = remove_stop_words(instance)
    words = instance[0]

    # 使用 nltk.collocations 的 BigramCollocationFinder
    bigram_finder = BigramCollocationFinder.from_words(words)
    # 2grams 按词频降序排列，前 400 个作为关键短语抽取
    bigrams = bigram_finder.nbest(BigramAssocMeasures.raw_freq, 400)
    for bigram in bigrams:
        feature_set[bigram] = 1
    return feature_set, instance[1]

def build_model(features):
    """
    用给定特征集构建朴素贝叶斯模型（NLTK 的朴素贝叶斯分类器）
    """
    model = nltk.NaiveBayesClassifier.train(features)
    return model

def probe_model(model, features, dataset_type='Train'):
    """
    计算测试集准确率，nltk 新版里已经没有 nltk.classify.accuracy()方法，
    所以这里自己编写 precision 值
    """
    right_cnt = 0
    sum_cnt = 0

    for feature in features:
        if model.classify(feature[0]) == feature[1]:
            right_cnt += 1
        sum_cnt += 1

    if sum_cnt > 0:
        accuracy = right_cnt * 100.0 / sum_cnt
        print("\n" + dataset_type + " Accuracy = %0.2f" % accuracy + "%")

def show_features(model, no_features=5):
    """
    显示对分类有帮助的特征（NLTK 中显示显著特征的方法）
    """
    print("\nFeature Importance")
    print("==================\n")
```

```
    print(model.show_most_informative_features(no_features))

def build_model_cycle_1(train_data, dev_data):
    """
    用 build_word_features 构建特征训练模型
    """
    # Build features for training set
    train_features = map(build_word_features, train_data)
    # Build features for test set
    dev_features = map(build_word_features, dev_data)
    # Build model
    model = build_model(train_features)
    # Look at the model Python3 的 map 返回的是迭代器，而不是列表，所以在训练使用后想再使用，需要再调用一
次
    train_features = map(build_word_features, train_data)
    print("\n 词语特征训练集准确率", end='')
    probe_model(model, train_features)
    print("词语特征验证集准确率", end='')
    probe_model(model, dev_features, 'Dev')
    return model

def build_model_cycle_2(train_data, dev_data):
    """
    用 build_negate_features 构建特征训练模型
    """

    # Build features for training set
    train_features = map(build_negate_features,train_data)
    # Build features for test set
    dev_features = map(build_negate_features,dev_data)
    # Build model
    model = build_model(train_features)
    # Look at the model
    train_features = map(build_negate_features, train_data)
    print("\n 否定词修饰特征训练集准确率", end='')
    probe_model(model, train_features)
    print("否定词修饰特征验证集准确率", end='')
    probe_model(model, dev_features,'Dev')

    return model


def build_model_cycle_3(train_data, dev_data):
    """
    用 build_keyphrase_features 构建特征训练模型
    """

    # Build features for training set
    train_features = map(build_keyphrase_features, train_data)
    # Build features for test set
    dev_features = map(build_keyphrase_features, dev_data)
    # Build model
    model = build_model(train_features)
    # Look at the model
    train_features = map(build_keyphrase_features, train_data)
    print("\n2gram 特征训练集准确率", end='')
    probe_model(model, train_features)
    print("2gram 特征验证集准确率", end='')
    probe_model(model, dev_features, 'Dev')
    test_features = map(build_keyphrase_features, test_data)
    print("2gram 特征测试集准确率", end='')
    probe_model(model, test_features, 'Test')
    return model

if __name__ == "__main__":

    # Load data
    input_dataset, y_labels = get_data()
    # Train data
    train_data, all_test_data, train_y, all_test_y = get_train_test(input_dataset, y_labels)
    # Dev data
    dev_data, test_data, dev_y, test_y = get_train_test(all_test_data, all_test_y)

    # Let us look at the data size in our different
    # datasets
    print("\nOriginal  Data Size   =", len(input_dataset))
    print("\nTraining  Data Size   =", len(train_data))
    print("\nDev       Data Size   =", len(dev_data))
```

```
    print("\nTesting   Data Size   =", len(test_data))

    # 用词语特征训练验证模型
    model_cycle_1 = build_model_cycle_1(train_data, dev_data)
    # 打印显著特征
    print("词语显著特征", end='')
    show_features(model_cycle_1)
    # 用否定词修饰的词语特征训练验证模型
    model_cycle_2 = build_model_cycle_2(train_data, dev_data)
    # 打印显著特征
    print("否定词修饰显著特征", end='')
    show_features(model_cycle_2)
    # 用 2gram 搭配特征训练验证模型
    model_cycle_3 = build_model_cycle_3(train_data, dev_data)
    # 打印显著特征
    print("2gram 显著特征", end='')
    show_features(model_cycle_3)
```

1.5.3 英文垃圾邮件分类

```
# coding:utf-8
"""
朴素贝叶斯垃圾邮件检测
"""
import numpy as np
import csv
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import csv
import codecs

# 文本预处理
def preprocessing(text):
    # 分词
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)]
    # 去除停用词
    stop = stopwords.words('english')
    tokens = [token for token in tokens if token not in stop]
    # 移除少于 3 个字母的单词
    tokens = [word for word in tokens if len(word) >= 3]
    # 大写字母转小写
    tokens = [word.lower() for word in tokens]
    # 词干还原
    lmtzr = WordNetLemmatizer()
    tokens = [lmtzr.lemmatize(word) for word in tokens]
    preprocessed_text = ' '.join(tokens)

    return preprocessed_text

def modelbuilding(sms_data, sms_labels):
    """
    构建分类器的流水线示例
    1. 构建训练集和测试集
    2. TFIDF 向量化器
    3. 构建朴素贝叶斯模型
    4. 打印准确率和其他评测方法
    5. 打印最相关特征
    :param sms_data:
    :param sms_labels:
    :return:
    """
    # 构建训练集和测试集步骤
    trainset_size = int(round(len(sms_data) * 0.70))
    # 我选择 70：30 的比例
    print('训练集大小：  ' + str(trainset_size) + '\n')
    x_train = np.array([''.join(el) for el in sms_data[0:trainset_size]])
    y_train = np.array([el for el in sms_labels[0:trainset_size]])
    x_test = np.array([''.join(el) for el in sms_data[trainset_size + 1:len(sms_data)]])
    y_test = np.array([el for el in sms_labels[trainset_size + 1:len(sms_labels)]])

    # We are building a TFIDF vectorizer here
    from sklearn.feature_extraction.text import TfidfVectorizer
```

```
    vectorizer = TfidfVectorizer(min_df=2, ngram_range=(1, 2), stop_words='english',
strip_accents='unicode', norm='l2')
    X_train = vectorizer.fit_transform(x_train)
    X_test = vectorizer.transform(x_test)

    from sklearn.naive_bayes import MultinomialNB
    clf = MultinomialNB().fit(X_train, y_train)
    y_nb_predicted = clf.predict(X_test)
    print(y_nb_predicted)

    # 输出测试集上的混淆矩阵
    from sklearn.metrics import confusion_matrix
    print(' \n 混淆矩阵 \n ')
    cm = confusion_matrix(y_test, y_nb_predicted)
    print(cm)

    # 输出测试集上的分类结果报告
    from sklearn.metrics import classification_report
    print('\n 分类报告')
    print(classification_report(y_test, y_nb_predicted))

    # 输出正负类的前 10 重要特征
    print("正负类前 10 重要特征")
    coefs = clf.coef_
    intercept = clf.intercept_
    feature_names = vectorizer.get_feature_names()
    coefs_with_fns = sorted(zip(clf.coef_[0], feature_names))
    n = 10
    top = zip(coefs_with_fns[:n], coefs_with_fns[:-(n + 1):-1])
    for (coef_1, fn_1), (coef_2, fn_2) in top:
        print('\t%.4f\t%-15s\t\t%.4f\t%-15s' % (coef_1, fn_1, coef_2, fn_2))

if __name__ == '__main__':
    sms_data = []
    sms_labels = []
    with codecs.open('data/SMSSpamCollection', 'rb', 'utf8', 'ignore') as infile:
        csv_reader = csv.reader(infile, delimiter='\t')
        for line in csv_reader:
            # 添加类别标记
            sms_labels.append(line[0])
            # 添加预处理后的文本
            sms_data.append(preprocessing(line[1]))
    print("原始数据大小：", len(sms_data))
    print("原始标签大小：", len(sms_labels))

    # 我们正则调用 model 构建函数
    modelbuilding(sms_data, sms_labels)
```

2. 聚类技术

与文本相关的另一个问题簇系是无监督式分类问题。关于这类问题，最常见的一种问题描述是“我手里有数以百万计的（非结构化）文档，是否能找到一种方式将它们分组，以便赋予其有意义的类别？”在这里，需要使用无监督的方式来对这些文本文档进行分组。文本聚类法（有时也叫聚类法）是目前最为常见的无监督式分组方式之一。

从历年论文文献上来看，基本没有介绍文本聚类技术的研究综述类文献，说明文本聚类技术本身并没有太多可作文章的点，或者说它本身并不复杂。

2.1 数据规范化

文本聚类中数据的处理的一个重要内容就是将数据进行归一化处理。所谓归一化，就是将原始数据矩阵中的每个数值，按照某种特定的运算法则把它变成为一个新值。常用的数据归一化方式有中心变换、极差正规化变换、标准差化变换、标准化变换。进行数据归一化的原因是避免具有较大值域的特征左右计算（相似度或分类）结果。例如考虑使用年龄和收入两个变量对人进行聚类。对于任意两个人，收入之差的绝对值多半比年龄之差的绝对值大很多。如果没有考虑到年龄和收入值域的差别，则对人的比较将被收入之差所左右。

2.1.1 中心化变换

中心化变换是对矩阵进行坐标轴平移处理方法。先求出每列的平均值，再将每列的各行数据都减去该列的平均值。

$$X_{ij}^{*} = X_{ij} - \bar{X}_j \quad i = 1,2,\dots,n; j = 1,2,\dots,p$$

其中，$\bar{X}_j = \frac{1}{n}\sum_{i=1}^{n} X_{ij}$，$X_{ij}$为矩阵中的原始值。

2.1.2 极差正规化变换

极差变换是找出每一个特征在所有文本中的最大值和最小值，这两者之差称为极差，然后使用每一个特征的每一个原始数据减去该特征分布的极小值，再除以极差，就得到变换后的数据。经过变换后，数据矩阵中每列即每个特征的最大数值为 1，最小数值为 0，其余数据取值均为 0～1；并且变换后的数据都不再具有量纲，即不再具有物理意义。

$$X_{ij}^{*}=\frac{X_{ij}-\min(X_{ij})}{R_j}\quad i=1,2,\dots,n;j=1,2,\dots,p$$

其中，$R_j=\max(X_{ij})-\min(X_{ij})$，$i=1,2,\dots n$。

2.1.3 极差标准化变换

极差标准变换同样先求出每个特征在所有文本中的极差，然后使用每个特征的每一个原始数据减去该特征分布中的平均值，再除以极差，就得到变换后的数据。经变换后，各个特征的均值为 0，极差均为 1，数据也不再具有量纲。

$$X_{ij}^{*}=\frac{X_{ij}-\overline{X_j}}{R_j}\quad i=1,2,\dots,n;j=1,2,\dots,p$$

其中，$R_j=\max(X_{ij})-\min(X_{ij})$，$i=1,2,\dots n$，$\overline{X_j}=\frac{1}{n}\sum_{i=1}^{n}X_{ij}$。

2.1.4 标准化变换

标准化变换要求先求出每个特征在所有文本中的均值及标准差，然后使用该特征的原始数据减去均值，并除以标准差。经过标准化变换处理后，每个特征即数据矩阵中每列数据的平均值为 0，方差为 1，且也不再具有量纲。

$$X_{ij}^{*}=\frac{X_{ij}-\overline{X_j}}{S_j}\quad i=1,2,\dots,n;j=1,2,\dots,p$$

其中，$S_j=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_{ij}-\bar{X}_j)^2}$，$\bar{X}_j=\frac{1}{n}\sum_{i=1}^{n}X_{ij}$。

2.1.5 向量归一化

向量归一化是将每个对象的 p 个特征视为该对象在 p 维向量空间中的值，使用原始数据除以该对象 p 个特征值的平方和和开方后的值，即为归一化后的数据。中心化变换、极差正规化变换、极差标准化变换与标准化变换都是从特征出发，对数据进行变换。向量归一化并不是从特征出发，而是从文本出发，对文本中的各个特征值进行归一化处理。将原来由若干个特征组成的空间向量转换为单位向量（变换后数据矩阵中的每行都是单位向量），即每个特征值均变换为在(0,1)之间。

$$X_{ij}^{*}=\frac{X_{ij}}{\sqrt{\sum_{i=1}^{p}X_{ij}^{2}}}\quad i=1,2,\dots,n;j=1,2,\dots,p$$

实践中使用最多的是向量归一化，因为其他 4 种方法都是对整个文本集按特征变换，在内存空间有限的情况下，当文本数量很大或者词语特征很多时，不可能将整个文本集读入内存，再进行变换。但是向量归一化是对单一文本进行的，因此可以只加载一篇文本向量便可以进行归一化处理。

示例

```
In [1]: import numpy as np
In [2]: a = np.array([0,3,4])
In [3]: np.linalg.norm(a)
Out[3]: 5.0
In [4]: a*(1.0/np.linalg.norm(a))
Out[4]: array([0. , 0.6, 0.8])
```

2.2 相似性度量

2.2.1 明氏距离

明氏距离不是一种距离，而是一组距离，它的定义式为

$$d_{ij}=\sqrt[p]{\sum_{k=1}^{n}|x_{ik}-x_{jk}|^{p}}$$

当 p=1 时，就是曼哈顿距离，当 p=2 时，就是欧式距离，当 p->∞时，就是切比雪夫距离。明氏距离有两个缺陷（1）将各个分量的量纲也就是单位，当作相同的看待。（2）没有考虑各个分量的分布（期望、方差等）可能是不同的。

2.2.2 曼哈顿距离

$$d_{ij} = \sum_{k=1}^{n} |x_{ik} - x_{jk}|$$

示例

```
In [9]: vector1
Out[9]: array([1, 2, 3])
In [10]: vector2
Out[10]: array([4, 5, 6])
In [6]: np.linalg.norm(vector1-vector2, ord=1)
Out[6]: 9.0
```

2.2.3 欧氏距离

$$d_{ij} = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

示例

```
In [9]: vector1
Out[9]: array([1, 2, 3])
In [10]: vector2
Out[10]: array([4, 5, 6])
In [7]: np.linalg.norm(vector1-vector2)
Out[7]: 5.196152422706632
```

2.2.4 切比雪夫距离

$$d_{ij} = \max_{k}(|x_{ik} - y_{ik}|)$$

示例

```
In [9]: vector1
Out[9]: array([1, 2, 3])
In [12]: vector2
Out[12]: array([4, 7, 5])
In [13]: np.linalg.norm(vector1-vector2, ord=np.inf)
Out[13]: 5.0
```

2.2.5 相关系数

$$\rho_{AB} = \frac{cov(A,B)}{\sqrt{D(A)}\sqrt{D(B)}}$$

相关系数是衡量随机变量 A 与 B 相关程度的一种方法，相关系数的取值范围是[-1,1]。相关系数的绝对值越大，则表明 A 与 B 相关程度越高。当 A 与 B 线性相关时，相关系数取值为 1（正线性相关）或-1（负线性相关）。

示例

```
In [23]: vector1
Out[23]: array([1, 2, 4])
In [24]: vector2
Out[24]: array([-2, -4, -8])
In [22]: np.corrcoef(vector1,vector2)
Out[22]:
array([[ 1., -1.],
   [-1.,  1.]])
In [26]: vector1
Out[26]: array([1, 2, 4])
In [27]: vector2
Out[27]: array([2, 4, 8])
In [28]: np.corrcoef(vector1, vector2)
Out[28]:
   array([[1., 1.],
          [1., 1.]])
```

2.2.6 余弦相似度

几何中夹角余弦可用来衡量两个向量方向的差异，借用这一概念来衡量样本向量之间的差异。定义为

$$\cos\theta = \frac{AB}{|A||B|}$$

示例

```
In [14]: vector1
Out[14]: array([1, 2, 3])
In [15]: vector2
Out[15]: array([4, 7, 5])
In [16]: np.dot(vector1,vector2)/(np.linalg.norm(vector1)*np.linalg.norm(vector2
    ...: ))
Out[16]: 0.9296696802013682
```

2.3 基本算法

2.3.1 K-Means

该方法非常直观，从其名称就可以看出它需要试着找出 k 组围绕着若干数据点的平均值。因此，该算法首先要拾取一些数据点来充当所有数据点的中心。接下来，该算法会将所有数据点各自分配给离其最近的那个中心。在这过程中，每完成一次迭代，其中心就要重新计算一次，然后继续迭代，直到达到中心不再变化的状态（即达到算法饱和）。

算法过程描述如下

（1） 先随机选取 K 个元组作为中心。

（2） 计算每个实例点到中心的距离（欧式距离）K*实例数量次，分配每个实例点到最近的中心。

（3） 更新中心。每个簇中所有实例点各维度的均值。

（4） 重复（2）（3）直到中心不再变化或迭代次数已到达。

K-Means 算法的缺陷有

（1） 对 k 个初始质心的选择比较敏感，容易陷入局部最小值。例如算法运行多次，有可能会得到不同的结果。解决该问题的方法是，使用多次的随机初始化，计算每一次建模得到的代价函数的值，选取代价函数最小结果作为聚类结果。代价函数为：

$$\text{cost} = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \mu_{c(i)}\right\|^2$$

其中 m 表示样本的数量，$x^{(i)}$表示样本 i，$\mu_{c(i)}$表示样本 i 所属簇的质心。

（2） k 的取值会影响聚类质量。此时可以使用肘部法则，也就是说选择不同的 k 值进行聚类并记录代价，最后选择处于肘部的代价值所对应的 k 值作为簇的个数。当肘部不存在时，需要根据经验进行选择。

（3） 非球形簇无法使用 K-Means 算法，此时可以使用密度聚类。

2.3.2 MiniBatches

Mini Batch K-Means 算法是 K-Means 算法的变种，采用小批量的数据子集减小计算时间。这里所谓的小批量是指每次训练算法时所随机抽取的数据子集，采用这些随机产生的子集进行算法训练，大大减小了计算时间，结果一般只略差于标准算法。K-Means 算法一般适用于已知簇数量，数据量小于 1 万条的情境，Mini Batch K-Means 算法适用于已知簇数量，数据量大于等于 1 万条的情境。Mini Batch K-Means 算法的迭代步骤是

（1） 从数据集中随机抽取一些数据形成小批量，把他们分配给最近的质心。

（2） 更新质心，与 K-Means 相比，数据的更新是在每一个小的样本集上。

2.3.3 single-pass

话题发现与跟踪（topic detection and tracking, TDT）的评测中常用的聚类方法是 single-pass 聚类，其原理简单、计算速度快，然而该算法的缺点也很明显：受输入顺序的影响，且聚类结果精度差。single-pass 聚类的基本流程如下：

（1） 接收一篇互联网文本向量 d；

（2） d 逐一与已有的话题中各报道进行相似度计算，并取最大者作为与该话题的相似度（single-link 策略）；

（3） 在所有话题间选出与 d 相似度最大的一个，以及此时的相似度值；

（4） 如果相似度大于阈值 TC，d 所对应的互联网文本被分配给这个话题模型文本类，跳至（6）；

（5） 如果相似度值小于阈值 TC，d 所对应的文本不属于已有的话题，创建新话题，同时把这篇文本归属创建的新话题模型文本类；

（6） 本次聚类结束，等待文本到来。

有两篇 single-pass 算法改进的文章。殷风景 2011 年提出了 ICIT 算法。改进方面有（1）词频统计针对具有实际意义的名词和动词，避免文本向量维度太高。（2）两篇文本的相似度=0.7*标题相似度+0.3*正文相似度，考虑了标题对于文本主题的概括性。（3）到达数据按代添加到聚类过程中，每一代包含 200 条数据，先在本代成员之间进行初步的相似度比较和聚类，再将这些初步类与已有话题进行比较和聚合，避免因数据到达顺序不同而使聚类结果有变化。（4）相似度计算采用了 average-link，准确度更佳，有效减少大类出现。（5）在当前代内完成聚类后加入一个比较调整的步骤，代内成员依次计算当前聚类结果下最相似的类簇是否就是自己所处的簇，不是则调整。陶舒怡 2014 年提出基于簇相合性的文本增量聚类算法。该算法的改进措施有（1）基于词项语义相似度的文本表示模型。（2）计算新增文本与已有簇的相合性实现增量聚类，它不仅计算了文本与簇的相似度，而且考虑了簇分布特征。（3）增量处理完成后对错分可能性大的文本重新指派类别。

2.4 简单示例

2.4.1 K-Means 示例

```
#coding:utf-8
"""
Kmeans 算法聚类文本示例
"""

import matplotlib.pyplot as plt
import numpy as np

# 加载文本数据
from time import time
from sklearn.datasets import load_files
print("loading documents ...")
t = time()
docs = load_files('datasets/clustering/data')
print("summary: {0} documents in {1} categories.".format(len(docs.data), len(docs.target_names)))
print("done in {0} seconds".format(time() - t))

# 文本向量化表示
from sklearn.feature_extraction.text import TfidfVectorizer
max_features = 20000
print("vectorizing documents ...")
t = time()
vectorizer = TfidfVectorizer(max_df=0.4, min_df=2, max_features=max_features, encoding='latin-1')
X = vectorizer.fit_transform((d for d in docs.data))
print("n_samples: %d, n_features: %d" % X.shape)
print("number of non-zero features in sample [{0}]: {1}".format(docs.filenames[0],
X[0].getnnz()))
print("done in {0} seconds".format(time() - t))

# 文本聚类
from sklearn.cluster import Kmeans, MiniBatchKMeans
print("clustering documents ...")
t = time()
n_clusters = 4
kmean = KMeans(n_clusters=n_clusters, max_iter=100, tol=0.01, verbose=1, n_init=3)
kmean.fit(X)
print("kmean: k={}, cost={}".format(n_clusters, int(kmean.inertia_)))
print("done in {0} seconds".format(time() - t))

# 打印实例数量
print(len(kmean.labels_))

# 打印实例 1000 到 1009 的簇号
print(kmean.labels_[1000:1010])

# 打印实例 1000 到 1009 的文件名
print(docs.filenames[1000:1010])

# 打印每个簇的前 10 个显著特征
print("Top terms per cluster:")
order_centroids = kmean.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names()
for i in range(n_clusters):
    print("Cluster %d:" % i, end='')
    for ind in order_centroids[i, :10]:
        print(' %s' % terms[ind], end='')
    print()
```

2.4.2 single-pass 示例

```
#coding:utf-8
"""
single-pass 增量聚类演示
"""

import numpy as np
from sklearn.datasets import load_files
from pyhanlp import *
import re
import codecs

NotionalTokenizer = JClass("com.hankcs.hanlp.tokenizer.NotionalTokenizer")
# 以文本在文本集中的顺序列出的文本向量矩阵（用 300 维向量表示）
text_vec = None
# 以文本在文本集中的顺序列出的话题序号列表
topic_serial = None
# 话题数量
topic_cnt = 0

# 加载词语向量词典
word_dict = dict()
with codecs.open('dictionary/cc.zh.300.vec', 'rb', 'utf-8', 'ignore') as infile:
    infile.readline()
    for line in infile:
        line = line.strip()
        if line:
            items_li = line.split()
            word = items_li[0]
            word_vec = np.array([float(w) for w in items_li[1::]])
            word_dict[word] = word_vec
print("load cc.zh.300.vec len = %d" % len(word_dict))

# 仅保留中文字符
def translate(text):
    p2 = re.compile(u'[^\u4e00-\u9fa5]')  # 中文的编码范围是：\u4e00 到\u9fa5
    zh = " ".join(p2.split(text)).strip()
    zh = ",".join(zh.split())
    res_str = zh  # 经过相关处理后得到中文的文本
    return res_str

# 预处理，实词分词器分词，查询词语向量，并返回文本向量
def preprocess(text):
    sen_vec = np.zeros((1, 300))
    # 去掉非中文字符
    text = translate(text)
    # 将\r\n 替换为空格
    text = re.sub(u'[\r\n]+', u' ', text)
    # 分词与词性标注，使用实词分词器
    word_li = NotionalTokenizer.segment(text)
    word_li = [w.word for w in word_li]
    # 去掉单字词
    word_li = [w for w in word_li if len(w)>1]
    # 查询每个词语的 fasttext 向量，计算句子向量
    valid_word_cnt = 0
    for word in word_li:
        if word in word_dict:
            sen_vec += word_dict[word]
            valid_word_cnt += 1
    if valid_word_cnt > 0:
        sen_vec = sen_vec*(1.0/valid_word_cnt)
    # 单位化句子向量
    sen_vec = sen_vec*(1.0/np.linalg.norm(sen_vec))
    return text, sen_vec

# single-pass
def single_pass(sen_vec, sim_threshold):
    global text_vec
    global topic_serial
    global topic_cnt
    if topic_cnt == 0:  # 第 1 次送入的文本
        # 添加文本向量
        text_vec = sen_vec
        # 话题数量+1
        topic_cnt += 1
```

```
        # 分配话题编号，话题编号从 1 开始
        topic_serial = [topic_cnt]
    else:  # 第 2 次及之后送入的文本
        # 文本逐一与已有的话题中的各文本进行相似度计算
        sim_vec = np.dot(sen_vec, text_vec.T)
        # 获取最大相似度值
        max_value = np.max(sim_vec)
        # 获取最大相似度值的文本所对应的话题编号
        topic_ser = topic_serial[np.argmax(sim_vec)]
        print("topic_ser", topic_ser, "max_value", max_value)
        # 添加文本向量
        text_vec = np.vstack([text_vec, sen_vec])
        # 分配话题编号
        if max_value >= sim_threshold:
            # 将文本聚合到该最大相似度的话题中
            topic_serial.append(topic_ser)
        else:
            # 否则新建话题，将文本聚合到该话题中
            # 话题数量+1
            topic_cnt += 1
            # 将新增的话题编号（也就是增加话题后的话题数量）分配给当前文本
            topic_serial.append(topic_cnt)

def main():
    # 加载数据
    data_all = load_files(container_path=r'data/news', categories=u'Sports',
                          encoding=u'gbk', decode_error=u'ignore')
    # 获取文本数据集
    data = data_all.data
    # 预处理后的文本数据集
    preprocessed_data = []
    # 进行增量聚类
    for text in data:
        text, text_vec = preprocess(text)
        single_pass(text_vec, 0.9)
        preprocessed_data.append(text)
    # 输出聚类结果
    with open('res_single_pass.txt', 'wb') as outfile:
        sorted_text = sorted(zip(topic_serial, preprocessed_data), key=lambda x:x[0])
        for topic_ser, text in sorted_text:
            out_str = u'%d\t%s\n' % (topic_ser, text)
            outfile.write(out_str.encode('utf-8', 'ignore'))
    print("program finished")
    # 在 mac 下释放向量内存时间较长，可以直接 ctrl+c 强制退出程序

if __name__ == '__main__':
    main()
```