机器学习 周志华 chap 8、chap 9 知识点总结

目录 /CONTENTS

Chap 8

集成学习

- 个体与集成
- Boosting
- Bagging与随机森林
- 结合策略
- 多样性

Chap 9

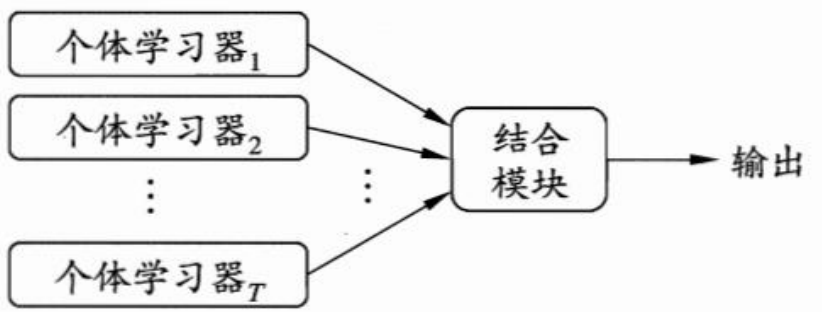
聚类

- 聚类任务的性能度量
- 距离计算
- 原型聚类
- 密度聚类
- 层次聚类

,

Chap 8 Part 1 个体与集成

- **集成学习**：集成学习通过构建并结合多个学习器完成学习任务，亦称多分类器系统。
- **集成学习的一般结构**：先产生一组“个体学习器”，再用某种策略将它们结合起来。



- **个体学习器**

个体学习器通常由一个现有的学习算法从训练数据产生。

- 同质集成：集成中只包含同种类型的个体学习器，其中的个体学习器亦称“基学习器”，其算法称为“基学习算法”；
- 异质集成：集成中同时包含不同类型的个体学习器，个体学习器由不同的学习算法生成。

,

Chap 8 Part 1 个体与集成

- **集成学习的优势**
  - 集成学习常可获得比单一学习器显著优越的**泛化性能**，对弱学习器尤为明显。

    （理论上使用弱学习器足以获得好的的性能，但实际任务中，往往使用比较强的学习器。

    注：弱学习器常指泛化性能略优于随机猜测的学习器，例如二分类问题精度略高于50%的分类器。）
  - 集成学习若想取得好的效果，个体学习器要“**好而不同**”。即个体学习器要有一定的准确性（至少不差于弱学习器）和多样性（学习器间具有差异）。

    核心问题 如何产生并结合“好而不同”的个体学习器。

    （注：假设基分类器的错误率相互独立，随着集成中个体分类器数目T的增大，集成的错误率将指数级下降，最终趋向于零。然而，相互独立不太可能，“准确性”和“多样性”本身就存在冲突。）

- **分类**（根据个体学习器的生成方式）：
  - 序列化方法：个体学习器间存在强依赖关系，可必须串行生成；

    代表：Boosting
  - 并行化方法：个体学习器间不存在强依赖关系，可同时生成。

    代表：Bagging、“随机森林”

- Boosting是一族可将弱学习器提升为强学习器的算法，最著名代表 AdaBoost。
- **工作机制**
  - 从初始训练集训练出一个基学习器；
  - 根据基学习器的表现对训练样本分布进行调整，使得先前基学习器做错的训练样本在后续得到更多关注；
  - 基于调整后的样本分布来训练下一个基学习器，直至基学习器数目达到事先指定的值T，将T个基学习器进行加权结合。
- **代表算法：AdaBoost**

  输入：训练数据集、弱学习算法

  输出：最终分类器
- **算法流程**
  - 初始化训练数据的权值分布；
  - 基于分布D_t从数据集D中训练出基分类器h_t；
  - 计算h_t在训练数据集上的分类误差率；
  - 确定分类器h_t的权重；
  - 更新训练数据集的权值分布；
  - 构建基本分类器的线性组合，得到最终的分类器。

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
基学习算法 $\mathfrak{L}$;
训练轮数 T.

过程：

1: $\mathcal{D}_1(\boldsymbol{x}) = 1/m.$
2: **for** $t = 1, 2, \ldots, T$ **do**
3: $\quad h_t = \mathfrak{L}(D, \mathcal{D}_t);$
4: $\quad \epsilon_t = P_{\boldsymbol{x} \sim \mathcal{D}_t}(h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}));$
5: $\quad$ **if** $\epsilon_t > 0.5$ **then break**
6: $\quad \alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right);$
7: $\quad \mathcal{D}_{t+1}(\boldsymbol{x}) = \frac{\mathcal{D}_t(\boldsymbol{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(\boldsymbol{x}) = f(\boldsymbol{x}) \\ \exp(\alpha_t), & \text{if } h_t(\boldsymbol{x}) \neq f(\boldsymbol{x}) \end{cases}$
$= \frac{\mathcal{D}_t(\boldsymbol{x}) \exp(-\alpha_t f(\boldsymbol{x}) h_t(\boldsymbol{x}))}{Z_t}$
8: **end for**

输出： $H(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x}) \right)$

图 8.3 AdaBoost算法

- Boosting要求基学习器能对特定的数据分布进行学习；
- **方法**
  - 重赋权法：在训练过程的每一轮中，根据样本分布为每个训练样本重新赋予一个权；
  - 重采样法：对于无法接受带权样本的基本分类器，可以通过“重采样法”处理。即每轮学习中，通过样本分布重新采样，得到新的样本集进行训练。

    优点：可获得重启动机会，避免未达到初始设置的学习轮数训练过程过早停止。
- Boosting在每轮都要检查当前生成的基分类器是否比随机猜测好（错误率是否小于0.5）。如果不满足，基学习器被抛弃，学习停止。这时已训练好的基分类器数目可能没有达到设置的学习轮数T；可以采用“重采样法”，重新采样训练基学习器，使学习过程得以持续。
- Boosting主要关注于降低前一轮训练的偏差。

- **核心问题**

欲得到泛化性能较强的集成，个体学习器应尽可能性能好并且差异大。对训练样本采样可以增大个体差异；同时个体学习器的训练子集不能太小，因此考虑使用相互有交叠的采样子集。

- **解决办法**

Bagging是并行式集成学习方法最著名的代表，基于自助采样法，每个分类器训练时，随机从原数据集中抽取相同数量的数据进行训练。这样采样子集间相互有交叠，但训练数据不至于太少。

- **基本流程**
  - 采样出 T 个含 m 个训练样本的采样集，然后基于每个采样集训练处一个基学习器，再将这些基学习器进行结合。

    （1）分类任务：简单投票法

    （2）回归任务：简单平均法
  - **优点**

    （1）能不经修改的用于多分类任务和回归任务。

    （2）每个基学习器只使用了初始训练集中约63.2%的样本，剩下的样本可用作验证集来对泛化性能进行包外估计。
  - **应用**

    Bagging主要关注降低方差，因此在不剪枝决策树、神经网络等易受样本扰动的学习器上效用更明显。

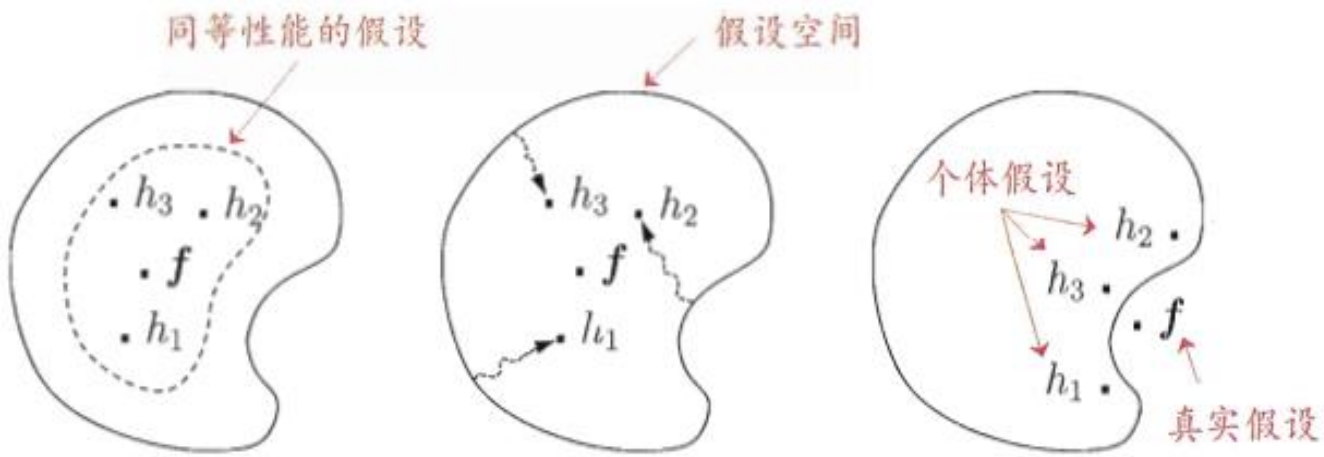
Chap 8 Part 3 随机森林

- 随机森林（RF）是Bagging的一个扩展变体，在以决策树为基学习器构建Bagging集成的基础上，进一步在决策树的训练过程中引入随机属性变量选择。
- **主要思想**

传统决策树选择当前节点属性集合中的最优属性；而随机森林先从该节点的属性集合中随机选择一部分特征，在这些随机选择的部分样本特征中选择一个最优的特征做决策树左右划分。

- **优点**
  - 多样性不仅来自样本扰动还来自属性扰动，泛化性能进一步提升
  - 随着学习器数目的增加，随机森林通常会收敛得到更低的泛化误差
  - 随机森林的训练效率常优于Bagging。

- **学习器结合的好处**
  - 减少因单学习器误选而导致的泛化性能不佳的风险；
  - 避免陷入糟糕的局部极小；
  - 扩大假设空间，可能学的更好的近似。



- **常见结合策略**

➢ **平均法：数值型输出**

（1）简单平均法

适用范围：个体学习器性能相近时

（2）加权平均法

适用范围：个体学习器性能相差较大

注：权重从训练数据中学习而得；

贝叶斯模型平均（BMA）基于后验概率来为不同模型赋予权重，可视为加权平均法的一种特殊实现。

➢ **投票法：分类任务**

（1）绝对多数投票法

描述：若某标记得票过半数，则预测为改标记，否则拒绝预测。

优点：提供了“拒绝预测”选项，提高了预测结果的可靠性。

（2）相对多数投票法

描述：预测为得票最多的标记，若同时又多个标记获得最高票，则从中选取一个。

（3）加权投票法

➢ **个体学习器输出值的类型**

类标记：硬投票

类概率：软投票

注：基于类概率的结合往往比基于类标记进行结合性能更好；

若基学习器的类型不同，则其类概率值不能直接进行比较；此时通常可将类概率转化为类标记输出再投票。

- **常见结合策略**

➢ **学习法：训练数据很多**

思想：通过另一个学习器来结合，将个体学习器称为初级学习器，用于结合的学习器称为次级学习器或元学习器。

典型代表：Stacking

描述：先从初始数据集训练出初级学习器，然后“生成”一个新数据集用于训练次级学习器。初级学习器的输出当作样例输入特征，初始样本的标记当作样例标记。

➢ **算法流程**

输入：训练数据集 T，T个初级学习算法，1个次级学习算法；

（1）循环训练好T个初级学习器；

（2）用训练初级学习器未使用的样本产生次级学习器的训练样本，常用交叉验证或留一法；

（3）用次级训练集训练次级学习器。

（4）将初级学习器的输出类概率作为次级学习器的输入属性，用多响应线性回归(MLR)作为次级学习算法效果较好，在MLR中使用不同的属性效果更佳。

- **回归学习：误差-分歧分解**

E是集成的泛化误差，$\bar{E}$表示个体学习器泛化误差的加权均值，$\bar{A}$表示个体学习器的加权分歧值：

$$E = \bar{E} - \bar{A} .$$

上式表明：个体学习器准确性越高、多样性越大，则集成越好。

- **多样性度量：考虑个体分类器的两两相似性**

➢ **成对性多样性度量指标**

(1) 不合度量：两个学习器预测得不一样的概率

(2) κ − 统计量：$\kappa = \frac{p_1 - p_2}{1 - p_2}$

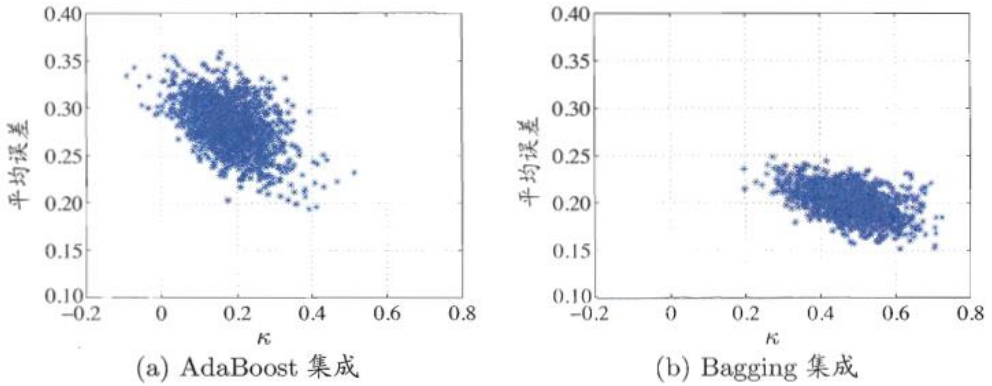


图 8.10 在 UCI 数据集 *tic-tac-toe* 上的 κ-误差图. 每个集成含 50 棵 C4.5 决策树

p_1是两个分类器取得一致的概率；p_2是两个分类器偶然达成一致的概率。若两个分类器完全一致，则$\kappa = 1$；若它们仅是偶然达成一致，则$\kappa = 0$。κ通常为非负值。

κ −误差图：将每一对分类器作为图上的一个点，横坐标是这对分类器的κ值，纵坐标是它们的平均误差。

- **增强多样性的方法**

方法名称	基本思路	代表方法	应用
数据样本扰动	给定初始数据集，可从中产生不同的数据子集，再利用不同的数据子集训练出不同的个体学习器	采样法	不稳定基学习器（训练样本稍加变化就会导致学习器有显著变动）：决策树、神经网络等
输入属性扰动	训练样本通常由一组属性描述，不同的属性子集训练处的个体学习器不同。	随机子空间算法	冗余属性多的数据
输出表示扰动	对输出表示进行操纵以增强多样性。	（1）翻转法：对训练样本的类标记稍作变动；（2）输出调制法：对输出表示进行转化；（3）ECOC法：将原任务拆解为多个可同时求解的子任务。	
算法参数扰动	通过随机设置不同的参数，产生差别较大的个体学习器	负相关法：显式地通过正则化项来强制个体神经网络使用不同的参数。	神经网络的隐层神经元数，初始连接权值等

Chap 9 Part 1 聚类任务的性能度量

- 聚类任务是无监督学习的一个重要应用。
- 聚类试图将数据集中的样本划分为若干个通常是不相交的子集，每个子集成为一个“簇”。
- **聚类目标**：聚类结果的“簇内相似度”高且“簇间相似度”低。
- **聚类性能度量指标**
  - 外部指标：将聚类结果与某个参考模型进行比较
  - 内部指标：直接考察聚类结果而不利用任何参考模型。

对数据集 $D=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_m\}$, 假定通过聚类给出的簇划分为 $\mathcal{C}=\{C_1,C_2,\ldots,C_k\}$, 参考模型给出的簇划分为 $\mathcal{C}^*=\{C_1^*,C_2^*,\ldots,C_s^*\}$. 相应地, 令 $\boldsymbol{\lambda}$ 与 $\boldsymbol{\lambda}^*$ 分别表示与 $\mathcal{C}$ 和 $\mathcal{C}^*$ 对应的簇标记向量. 我们将样本两两配对考虑, 定义

$$
\begin{aligned}
a &= |SS|, \quad SS=\{(\boldsymbol{x}_i,\boldsymbol{x}_j) \mid \lambda_i=\lambda_j, \lambda_i^*=\lambda_j^*, i<j)\},\\
b &= |SD|, \quad SD=\{(\boldsymbol{x}_i,\boldsymbol{x}_j) \mid \lambda_i=\lambda_j, \lambda_i^*\neq\lambda_j^*, i<j)\},\\
c &= |DS|, \quad DS=\{(\boldsymbol{x}_i,\boldsymbol{x}_j) \mid \lambda_i\neq\lambda_j, \lambda_i^*=\lambda_j^*, i<j)\},\\
d &= |DD|, \quad DD=\{(\boldsymbol{x}_i,\boldsymbol{x}_j) \mid \lambda_i\neq\lambda_j, \lambda_i^*\neq\lambda_j^*, i<j)\},
\end{aligned}
$$

其中集合 SS 包含了在 $\mathcal{C}$ 中隶属于相同簇且在 $\mathcal{C}^*$ 中也隶属于相同簇的样本对, 集合 SD 包含了在 $\mathcal{C}$ 中隶属于相同簇但在 $\mathcal{C}^*$ 中隶属于不同簇的样本对, …… 由于每个样本对 $(\boldsymbol{x}_i,\boldsymbol{x}_j)$ $(i<j)$ 仅能出现在一个集合中, 因此有 $a+b+c+d=m(m-1)/2$ 成立.

Chap 9 Part 1 聚类任务的性能度量

- **常用的外部指标**

- Jaccard 系数(Jaccard Coefficient, 简称 JC)

$$\mathrm{JC} = \frac{a}{a+b+c}.$$

- FM 指数(Fowlkes and Mallows Index, 简称 FMI)

$$\mathrm{FMI} = \sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}}.$$

- Rand 指数(Rand Index, 简称 RI)

$$\mathrm{RI} = \frac{2(a+d)}{m(m-1)}.$$

显然, 上述性能度量的结果值均在 $[0,1]$ 区间, 值越大越好.

- **常用的内部指标**

- DB 指数(Davies-Bouldin Index, 简称 DBI)

$$\mathrm{DBI} = \frac{1}{k}\sum_{i=1}^{k}\max_{j\neq i}\left(\frac{\mathrm{avg}(C_i)+\mathrm{avg}(C_j)}{d_{\mathrm{cen}}(\boldsymbol{\mu}_i,\boldsymbol{\mu}_j)}\right).$$

- Dunn 指数(Dunn Index, 简称 DI)

$$\mathrm{DI} = \min_{1\leqslant i\leqslant k}\left\{\min_{j\neq i}\left(\frac{d_{\min}(C_i,C_j)}{\max_{1\leqslant l\leqslant k}\mathrm{diam}(C_l)}\right)\right\}$$

显然, DBI 的值越小越好, 而 DI 则相反, 值越大越好.

- **距离越大，相似度越小**
  - 有序属性：可直接在属性上计算距离，闵可夫斯基距离计算；
  - 无须属性：不可直接在属性值上计算距离，可使用VDM距离；
  - 样本空间中不同属性重要性不同，加权距离。

- **闵可夫斯基距离**：可用于有序属性

$$\mathrm{dist}_{\mathrm{mk}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\sum_{u=1}^{n} |x_{iu} - x_{ju}|^p \right)^{\frac{1}{p}} .$$

p=2时，闵可夫斯基距离即欧氏距离；

p=1时，闵可夫斯基距离即曼哈顿距离。

- **VDM距离**

对无序属性可采用 VDM (Value Difference Metric) [Stanfill and Waltz, 1986]. 令 $m_{u,a}$ 表示在属性 u 上取值为 a 的样本数, $m_{u,a,i}$ 表示在第 i 个样本簇中在属性 u 上取值为 a 的样本数, k 为样本簇数, 则属性 u 上两个离散值 a 与 b 之间的 VDM 距离为

$$\mathrm{VDM}_p(a,b) = \sum_{i=1}^{k} \left| \frac{m_{u,a,i}}{m_{u,a}} - \frac{m_{u,b,i}}{m_{u,b}} \right|^p . \qquad (9.21)$$

注：用于相似度度量的距离不一定满足距离度量的所有性质；不满足直递性的距离称为非度量距离。

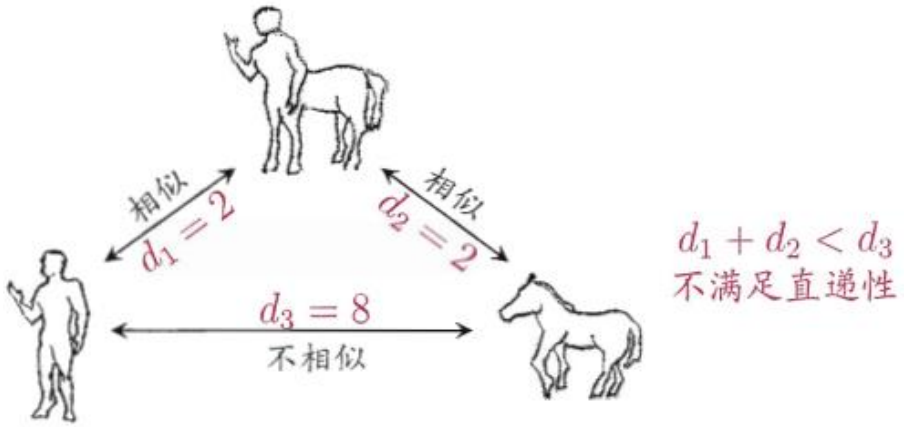


图 9.1 非度量距离的一个例子

- 原型聚类亦称“基于原型的聚类”，此类算法假设聚类结构能通过一组原型刻画。
- **算法思想**

  先对原型进行初始化，然后对原型进行迭代更新求解。
- ***K*均值算法（*k*-means)**
  - **思想**：对于给定的样本集，按照样本之间的距离大小，将样本集划分为K个簇。让簇内的点尽量紧密的连在一起，而让簇间的距离尽量的大

    给定样本集*D*，*k*-means算法针对聚类所得簇划分C最小化平方误差。

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} ||\boldsymbol{x} - \boldsymbol{\mu}_i||_2^2 ,$$

    上式在一定程度上刻画了簇内样本围绕簇均值向量的紧密程度，E值越小则簇内样本相似度越高。
  - **求解：***k*-means采用贪心策略，通过迭代优化来近似优化求解式。

- **K均值算法（k-means)**
  - **算法流程** 其中第一行对均值向量进行初始化，在第4-8行与第9-16行依次对当前簇划分及均值向量迭代更新，若迭代更新后聚类结果保持不变，则在第18行将当前簇划分结果返回。

输入: 样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
聚类簇数 k.
过程:
1: 从 D 中随机选择 k 个样本作为初始均值向量 $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k\}$
2: **repeat**
3: 令 $C_i = \varnothing \ (1 \leqslant i \leqslant k)$
4: **for** $j = 1, 2, \ldots, m$ **do**
5: 计算样本 $\boldsymbol{x}_j$ 与各均值向量 $\boldsymbol{\mu}_i \ (1 \leqslant i \leqslant k)$ 的距离: $d_{ji} = \|\boldsymbol{x}_j - \boldsymbol{\mu}_i\|_2$;
6: 根据距离最近的均值向量确定 $\boldsymbol{x}_j$ 的簇标记: $\lambda_j = \arg\min_{i \in \{1,2,\ldots,k\}} d_{ji}$;
7: 将样本 $\boldsymbol{x}_j$ 划入相应的簇: $C_{\lambda_j} = C_{\lambda_j} \bigcup \{\boldsymbol{x}_j\}$;
8: **end for**
9: **for** $i = 1, 2, \ldots, k$ **do**
10: 计算新均值向量: $\boldsymbol{\mu}'_i = \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$;
11: **if** $\boldsymbol{\mu}'_i \neq \boldsymbol{\mu}_i$ **then**
12: 将当前均值向量 $\boldsymbol{\mu}_i$ 更新为 $\boldsymbol{\mu}'_i$
13: **else**
14: 保持当前均值向量不变
15: **end if**
16: **end for**
17: **until** 当前均值向量均未更新
输出: 簇划分 $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$

图 9.2 k 均值算法

  - **优点**

    简单，易于理解和实现；收敛快，一般仅需5-10次迭代即可，高效
  - **缺点**

    （1）对K值得选取把握不同对结果有很大的不同；

    （2）对于初始点的选取敏感，不同的随机初始点得到的聚类结果可能完全不同；

    （3）对于不是凸的数据集比较难收敛；

    （4）结果不一定是全局最优，只能保证局部最优。

- **学习向量量化（LVQ）**

➢ **思想** 采样原型向量来刻画聚类结构。LVQ假设数据样本带有类别标价，学习过程利用样本的这些监督信息来辅助聚类。

➢ **目标** LVQ的目标是学的一组*n*维原型向量，每个原型向量代表一个聚类簇。

➢ **算法流程**

第1行先对原型向量进行初始化，第2~12行对原型向量进行迭代优化。在每一轮迭代中，算法随机选取一个有标记训练样本，找出与其距离最近的原型向量，并根据两者的类别标记是否一致来对原型向量进行相应的更新。第12行，若算法的停止条件已经满足，则将当前原型向量作为最终结果返回。

输入： 样本集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
原型向量个数 q, 各原型向量预设的类别标记 $\{t_1, t_2, \ldots, t_q\}$;
学习率 $\eta \in (0, 1)$.

过程：

1: 初始化一组原型向量 $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_q\}$
2: **repeat**
3: 从样本集 D 随机选取样本 $(\boldsymbol{x}_j, y_j)$;
4: 计算样本 $\boldsymbol{x}_j$ 与 $\boldsymbol{p}_i$ $(1 \leqslant i \leqslant q)$ 的距离: $d_{ji} = \|\boldsymbol{x}_j - \boldsymbol{p}_i\|_2$;
5: 找出与 $\boldsymbol{x}_j$ 距离最近的原型向量 p_{i^*}, $i^* = \arg\min_{i \in \{1,2,\ldots,q\}} d_{ji}$;
6: **if** $y_j = t_{i^*}$ **then**
7: $\boldsymbol{p}' = \boldsymbol{p}_{i^*} + \eta \cdot (\boldsymbol{x}_j - \boldsymbol{p}_{i^*})$
8: **else**
9: $\boldsymbol{p}' = \boldsymbol{p}_{i^*} - \eta \cdot (\boldsymbol{x}_j - \boldsymbol{p}_{i^*})$
10: **end if**
11: 将原型向量 $\boldsymbol{p}_{i^*}$ 更新为 $\boldsymbol{p}'$
12: **until** 满足停止条件

输出： 原型向量 $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_q\}$

图 9.4 学习向量量化算法

- **学习向量量化（LVQ）**

➢ **思想** 采样原型向量来刻画聚类结构。LVQ假设数据样本带有类别标价，学习过程利用样本的这些监督信息来辅助聚类。

➢ **目标** LVQ的目标是学的一组*n*维原型向量，每个原型向量代表一个聚类簇。

➢ **算法流程**

第1行先对圆形项链进行初始化，第2~12行对圆形项链进行迭代优化。在每一轮迭代中，算法随机选取一个有标记训练样本，找出与其距离最近的原型向量，并根据两者的类别标记是否一致来对原型向量进行相应的更新。第12行，若算法的停止条件已经满足，则将当前原型向量作为最终结果返回。

输入： 样本集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
原型向量个数 q, 各原型向量预设的类别标记 $\{t_1, t_2, \ldots, t_q\}$;
学习率 $\eta \in (0, 1)$.

过程：

1: 初始化一组原型向量 $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_q\}$
2: **repeat**
3: 从样本集 D 随机选取样本 $(\boldsymbol{x}_j, y_j)$;
4: 计算样本 $\boldsymbol{x}_j$ 与 $\boldsymbol{p}_i$ $(1 \leqslant i \leqslant q)$ 的距离: $d_{ji} = \|\boldsymbol{x}_j - \boldsymbol{p}_i\|_2$;
5: 找出与 $\boldsymbol{x}_j$ 距离最近的原型向量 p_{i^*}, $i^* = \arg\min_{i \in \{1,2,\ldots,q\}} d_{ji}$;
6: **if** $y_j = t_{i^*}$ **then**
7: $\boldsymbol{p}' = \boldsymbol{p}_{i^*} + \eta \cdot (\boldsymbol{x}_j - \boldsymbol{p}_{i^*})$
8: **else**
9: $\boldsymbol{p}' = \boldsymbol{p}_{i^*} - \eta \cdot (\boldsymbol{x}_j - \boldsymbol{p}_{i^*})$
10: **end if**
11: 将原型向量 $\boldsymbol{p}_{i^*}$ 更新为 $\boldsymbol{p}'$
12: **until** 满足停止条件

输出： 原型向量 $\{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_q\}$

图 9.4 学习向量量化算法

- **高斯混合聚类**

➢ **思想** 高斯混合聚类是采用概率模型（高斯分布）来表达聚类原型，簇划分则由原型对应后验概率确定。

我们可定义高斯混合分布

$$p_{\mathcal{M}}(\boldsymbol{x}) = \sum_{i=1}^{k} \alpha_i \cdot p(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) , \tag{9.29}$$

该分布共由 k 个混合成分组成, 每个混合成分对应一个高斯分布. 其中 $\boldsymbol{\mu}_i$ 与 $\boldsymbol{\Sigma}_i$ 是第 i 个高斯混合成分的参数, 而 $\alpha_i > 0$ 为相应的“混合系数”(mixture coefficient), $\sum_{i=1}^{k} \alpha_i = 1$.

➢ **难点** 求解模型参数 $\{(\alpha_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \mid 1 \leqslant i \leqslant k\}$

➢ **求解** 采用EM算法进行迭代优化求解：

E步：在每步迭代中，先根据当前参数来计算每个样本属于每个高斯成分的后验概率；

M步：更新模型参数

➢ **算法流程** 第1行对高斯混合分布的模型参数进行初始化，2~12行基于EM算法对模型参数进行迭代更新，若EM算法的停止条件满足则在第14-17行根据高斯混合分布确定簇划分，在第18行返回最终结果。

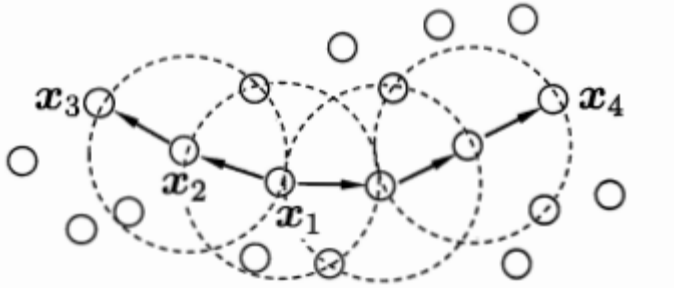
输入：样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
高斯混合成分个数 k.
过程：
1: 初始化高斯混合分布的模型参数 $\{(\alpha_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \mid 1 \leqslant i \leqslant k\}$
2: **repeat**
3: **for** $j = 1, 2, \ldots, m$ **do**
4: 根据式(9.30)计算 $\boldsymbol{x}_j$ 由各混合成分生成的后验概率, 即 $\gamma_{ji} = p_{\mathcal{M}}(z_j = i \mid \boldsymbol{x}_j)$ $(1 \leqslant i \leqslant k)$
5: **end for**
6: **for** $i = 1, 2, \ldots, k$ **do**
7: 计算新均值向量: $\boldsymbol{\mu}_i' = \frac{\sum_{j=1}^{m} \gamma_{ji} \boldsymbol{x}_j}{\sum_{j=1}^{m} \gamma_{ji}}$;
8: 计算新协方差矩阵: $\boldsymbol{\Sigma}_i' = \frac{\sum_{j=1}^{m} \gamma_{ji} (\boldsymbol{x}_j - \boldsymbol{\mu}_i')(\boldsymbol{x}_j - \boldsymbol{\mu}_i')^{\mathrm{T}}}{\sum_{j=1}^{m} \gamma_{ji}}$;
9: 计算新混合系数: $\alpha_i' = \frac{\sum_{j=1}^{m} \gamma_{ji}}{m}$;
10: **end for**
11: 将模型参数 $\{(\alpha_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \mid 1 \leqslant i \leqslant k\}$ 更新为 $\{(\alpha_i', \boldsymbol{\mu}_i', \boldsymbol{\Sigma}_i') \mid 1 \leqslant i \leqslant k\}$
12: **until** 满足停止条件
13: $C_i = \varnothing$ $(1 \leqslant i \leqslant k)$
14: **for** $j = 1, 2, \ldots, m$ **do**
15: 根据式(9.31)确定 $\boldsymbol{x}_j$ 的簇标记 λ_j;
16: 将 $\boldsymbol{x}_j$ 划入相应的簇: $C_{\lambda_j} = C_{\lambda_j} \bigcup \{\boldsymbol{x}_j\}$
17: **end for**
输出：簇划分 $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$

图 9.6 高斯混合聚类算法

- 密度聚类亦称基于密度的聚类，此类算法假设聚类结果能通过样本分布的紧密程度确定。
- **代表算法 DBSCAN**，基于一组邻域参数$(\epsilon, MinPts)$来刻画样本分布的紧密程度。

- **基础概念**

- ϵ-邻域: 对 $\boldsymbol{x}_j \in D$, 其 ϵ-邻域包含样本集 D 中与 $\boldsymbol{x}_j$ 的距离不大于 ϵ 的样本, 即 $N_\epsilon(\boldsymbol{x}_j) = \{\boldsymbol{x}_i \in D \mid \text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j) \leqslant \epsilon\}$;
- 核心对象(core object): 若 $\boldsymbol{x}_j$ 的 ϵ-邻域至少包含 $MinPts$ 个样本, 即 $|N_\epsilon(\boldsymbol{x}_j)| \geqslant MinPts$, 则 $\boldsymbol{x}_j$ 是一个核心对象;
- 密度直达(directly density-reachable): 若 $\boldsymbol{x}_j$ 位于 $\boldsymbol{x}_i$ 的 ϵ-邻域中, 且 $\boldsymbol{x}_i$ 是核心对象, 则称 $\boldsymbol{x}_j$ 由 $\boldsymbol{x}_i$ 密度直达;
- 密度可达(density-reachable): 对 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$, 若存在样本序列 $\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n$, 其中 $\boldsymbol{p}_1 = \boldsymbol{x}_i$, $\boldsymbol{p}_n = \boldsymbol{x}_j$ 且 $\boldsymbol{p}_{i+1}$ 由 $\boldsymbol{p}_i$ 密度直达, 则称 $\boldsymbol{x}_j$ 由 $\boldsymbol{x}_i$ 密度可达;
- 密度相连(density-connected): 对 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$, 若存在 $\boldsymbol{x}_k$ 使得 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$ 均由 $\boldsymbol{x}_k$ 密度可达, 则称 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$ 密度相连.



- **DBSAN 簇的定义**

基于这些概念, DBSCAN 将“簇”定义为: 由密度可达关系导出的最大的密度相连样本集合. 形式化地说, 给定邻域参数 $(\epsilon, MinPts)$, 簇 $C \subseteq D$ 是满足以下性质的非空样本子集:

$$\text{连接性(connectivity)}: \boldsymbol{x}_i \in C, \boldsymbol{x}_j \in C \Rightarrow \boldsymbol{x}_i \text{ 与 } \boldsymbol{x}_j \text{ 密度相连} \tag{9.39}$$

$$\text{最大性(maximality)}: \boldsymbol{x}_i \in C, \boldsymbol{x}_j \text{ 由 } \boldsymbol{x}_i \text{ 密度可达} \Rightarrow \boldsymbol{x}_j \in C \tag{9.40}$$

➢ **算法描述**

DBSCAN算法先任选数据集中的一个核心对象为种子，再由此出发确定相应的聚类簇。

➢ **算法流程**

在第1~7行中，算法先根据给定的邻域参数找出所有核心对象；10~24行，以任一核心对象为出发点，找出由其密度可达的样本生成聚类簇，直到所有核心对象均被访问过为止。

输入: 样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
　　邻域参数 $(\epsilon, MinPts)$.
过程:
1: 初始化核心对象集合: $\Omega = \varnothing$
2: **for** $j = 1, 2, \ldots, m$ **do**
3: 　确定样本 $\boldsymbol{x}_j$ 的 ϵ-邻域 $N_\epsilon(\boldsymbol{x}_j)$;
4: 　**if** $|N_\epsilon(\boldsymbol{x}_j)| \geqslant MinPts$ **then**
5: 　　将样本 $\boldsymbol{x}_j$ 加入核心对象集合: $\Omega = \Omega \bigcup \{\boldsymbol{x}_j\}$
6: 　**end if**
7: **end for**
8: 初始化聚类簇数: $k = 0$
9: 初始化未访问样本集合: $\Gamma = D$
10: **while** $\Omega \neq \varnothing$ **do**
11: 　记录当前未访问样本集合: $\Gamma_{\text{old}} = \Gamma$;
12: 　随机选取一个核心对象 $\boldsymbol{o} \in \Omega$, 初始化队列 $Q = < \boldsymbol{o} >$;
13: 　$\Gamma = \Gamma \setminus \{\boldsymbol{o}\}$;
14: 　**while** $Q \neq \varnothing$ **do**
15: 　　取出队列 Q 中的首个样本 $\boldsymbol{q}$;
16: 　　**if** $|N_\epsilon(\boldsymbol{q})| \geqslant MinPts$ **then**
17: 　　令 $\Delta = N_\epsilon(\boldsymbol{q}) \bigcap \Gamma$;
18: 　　将 Δ 中的样本加入队列 Q;
19: 　　$\Gamma = \Gamma \setminus \Delta$;
20: 　　**end if**
21: 　**end while**
22: 　$k = k + 1$, 生成聚类簇 $C_k = \Gamma_{\text{old}} \setminus \Gamma$;
23: 　$\Omega = \Omega \setminus C_k$
24: **end while**
输出: 簇划分 $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$

图 9.9　DBSCAN 算法

Chap 9 Part 5 层次聚类

- 层次聚类试图在不同层次对数据进行划分，从而形成树形的聚类结构，数据集的划分可采用自底向上的聚合策略，也可采用自顶向下的分拆策略。
- **代表算法** AGNES是一种采用自底向上聚合策略的层次聚类算法，它将数据集中的每个样本看作一个初始聚类簇，然后在算法运行的每一步中找出距离最近两个聚类簇进行合并，该过程不断重复，直至达到预设的聚类簇个数。
- **关键问题** 如何计算聚类簇之间的距离。
- **解决方法** 每个簇是一个样本集合，转化成集合的距离计算。

最小距离: $d_{\min}(C_i, C_j) = \min_{\boldsymbol{x} \in C_i, \boldsymbol{z} \in C_j} \operatorname{dist}(\boldsymbol{x}, \boldsymbol{z})$,

最大距离: $d_{\max}(C_i, C_j) = \max_{\boldsymbol{x} \in C_i, \boldsymbol{z} \in C_j} \operatorname{dist}(\boldsymbol{x}, \boldsymbol{z})$,

平均距离: $d_{\text{avg}}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{\boldsymbol{x} \in C_i} \sum_{\boldsymbol{z} \in C_j} \operatorname{dist}(\boldsymbol{x}, \boldsymbol{z})$.

- **算法流程**

AGNES 算法描述如图 9.11 所示. 在第 1–9 行, 算法先对仅含一个样本的初始聚类簇和相应的距离矩阵进行初始化; 然后在第 11–23 行, AGNES 不断合并距离最近的聚类簇, 并对合并得到的聚类簇的距离矩阵进行更新; 上述过程不断重复, 直至达到预设的聚类簇数.

输入: 样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
聚类簇距离度量函数 d;
聚类簇数 k.

过程:

1: **for** $j = 1, 2, \ldots, m$ **do**
2: $C_j = \{\boldsymbol{x}_j\}$
3: **end for**
4: **for** $i = 1, 2, \ldots, m$ **do**
5: **for** $j = 1, 2, \ldots, m$ **do**
6: $M(i, j) = d(C_i, C_j)$;
7: $M(j, i) = M(i, j)$
8: **end for**
9: **end for**
10: 设置当前聚类簇个数: $q = m$
11: **while** $q > k$ **do**
12: 找出距离最近的两个聚类簇 C_{i^*} 和 C_{j^*};
13: 合并 C_{i^*} 和 C_{j^*}: $C_{i^*} = C_{i^*} \bigcup C_{j^*}$;
14: **for** $j = j^* + 1, j^* + 2, \ldots, q$ **do**
15: 将聚类簇 C_j 重编号为 C_{j-1}
16: **end for**
17: 删除距离矩阵 M 的第 j^* 行与第 j^* 列;
18: **for** $j = 1, 2, \ldots, q - 1$ **do**
19: $M(i^*, j) = d(C_{i^*}, C_j)$;
20: $M(j, i^*) = M(i^*, j)$
21: **end for**
22: $q = q - 1$
23: **end while**

输出: 簇划分 $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$

图 9.11 AGNES 算法

《机器学习•周志华》

第十章 降维与度量学习

目录 /CONTENTS

-降维与相关算法

1.降维的概念

2.K近邻学习

3.主成分分析

降维

-降维的概念：

- 降维(Dimensionality Reduction,DR)是指采用线性或者非线性的映射方法将高维空间的样本映射到低维空间。降维相当于获得低维空间下的数据等价表示，实现高维数据的可视化呈现。

-为什么要对数据进行降维：

- 等价的低维数据更方便存储、 处理、 计算和使用。
- 降维能够去除数据噪声、 降低算法开销。

-降维的应用：

- 降维还可应用于文本分类和数据压缩等领域。
- 降维可以得到原始数据的简化表示以加速后续处理或者改进输出结果，因此它已经成为很多算法数据进行预处理的重要手段。

-降维的分类：

- 线性降维方法：如主成分分析（PCA）
- 非线性降维方法（流形学习方法）：如等距离映射算法ISOMAP

K近邻学习

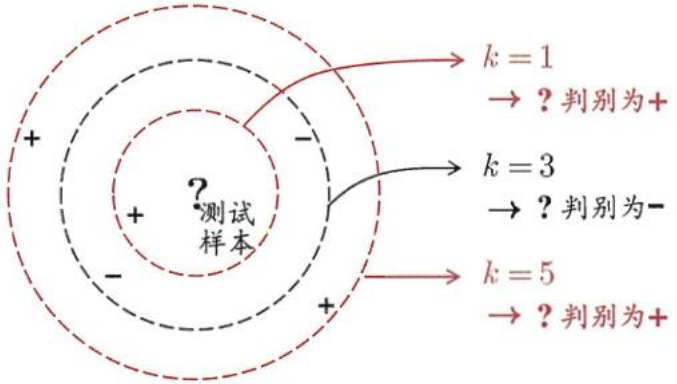
-k近邻学习的概念

- k近邻(k-Nearest Neighbor，简称kNN)学习是一种常用的监督学习方法，其工作机制非常简单：给定测试样本，基于某种距离度量找出训练集中与其最靠近的k个训练样本，然后基于这k个“邻居”的信息来进行预测。通常，在分类任务中可使用“投票法”，即选择这k个样本中出现最多的类别标记作为预测结果；在回归任务中可使用“平均法”，即将这k个样本的实值输出标记的平均值作为预测结果；还可基于距离远近进行加权平均或加权投票，距离越近的样本权重越大。

K近邻学习

-kNN算法计算流程如下：

- 计算每个样本点与测试点的距离；
- 排序后取距离值最小的k个样本点作为k-近邻；
- 获取k-近邻的分类标签并计算各分类出现的次数；
- 找出出现次数最多的分类,返回该值作为测试点的分类结果。



K近邻学习

-kNN算法关键：

- kNN算法本身简单有效，能处理大规模的数据分类，尤其适用于样本分类边界不明显的情况。
- kNN算法计算量较大且运行时需要大量内存：对每一个待分类数据都要计算它到全体已知样本的距离，才能求得它的k个最近邻点。
- 三个基本要素：k值的选择；距离度量方法；分类决策规则。

主成分分析

-主成分分析概念

- 主成分分析（Principal Component Analysis，简称PCA）是最常用的一种降维方法。
- 将原来较多的指标简化为少数几个新的综合指标的多元统计方法。
- 主成分分析中的主成分：由原始指标综合形成的几个新指标。依据主成分所含信息量的大小成为第一成分，第二成分等等。

主成分分析

-主成分分析算法思想：

- 通过某种线性投影，将高维的数据映射到低维的空间中表示,并期望在所投影的维度上数据的方差最大，以此使用较少的数据维度，同时保留住较多的原数据点的特性。
- PCA是将n维特征向量映射到r维上(r<n)，映射的过程要求每个维度的样本方差最大化，达到尽量使新的r维特征向量之间互不相关的目的。这些数据中拥有方差最大的r个维度被称为主成分。

主成分分析

-主成分分析算法流程：

输入： 样本集 $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$;
低维空间维数 d'.

过程：

1: 对所有样本进行中心化: $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i$;
2: 计算样本的协方差矩阵 $\mathbf{X}\mathbf{X}^{\mathrm{T}}$;
3: 对协方差矩阵 $\mathbf{X}\mathbf{X}^{\mathrm{T}}$ 做特征值分解;
4: 取最大的 d' 个特征值所对应的特征向量 $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_{d'}$.

输出： 投影矩阵 $\mathbf{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_{d'})$.

- 我们使用贡献率来判断选择主成分的个数。贡献率是指选取的d’个特征值总和占全部特征值总和的比重，一般应取85%以上。公式如下 $\frac{\sum_{i=1}^{d'} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \geqslant t$

- 应用场景： PCA是丢失原始数据信息最少的一种线性降维方式,可以将PCA应用在数据压缩、 数据可视化、提升机器学习速度等场景中。

Tensorflow基础知识

目录 /CONTENTS

Tensorflow

- 搭建
- 开发
- 可视化
- GPU
- 共享变量

Part 1 搭建（Python）

- **适用于单系统多用户的安装方式：基于VirtualEnv的安装**

使用 VirtualEnv 创建一个隔离的容器,来安装 TensorFlow。这是可选的, 但是这样做能使排查安装问题变得更容易。

首先, 安装所有必备工具:
$ sudo apt-get install python-pip python-dev python-virtualenv（用户环境已安装）
接下来, 建立一个全新的 virtualenv 环境。 为了将环境建立在设定的/tensorflow 目录下, 例如~/tensorflow执行:
$ virtualenv --system-site-packages ~/tensorflow
$ cd ~/tensorflow
然后, 激活 virtualenv:
$ source bin/activate # 如果使用 bash
(tensorflow)$ # 终端提示符应该发生变化
在 virtualenv 内, 安装 TensorFlow:
(tensorflow)$ pip install tensorflow-gpu==1.14.0

注意：目录~/tensorflow在个人用户下推荐换成/home/[用户名]/tensorflow

Part 1 搭建（Python）

- **适用于单系统多用户的安装方式：基于VirtualEnv的安装-版本关联**

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow-2.0.0	2.7, 3.3-3.7	GCC 7.3.1	Bazel 0.26.1	7.4	10.0
tensorflow_gpu-1.14.0	2.7, 3.3-3.7	GCC 4.8	Bazel 0.24.1	7.4	10.0
tensorflow_gpu-1.13.1	2.7, 3.3-3.7	GCC 4.8	Bazel 0.19.2	7.4	10.0
tensorflow_gpu-1.12.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.11.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.15.0	7	9
tensorflow_gpu-1.9.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.11.0	7	9
tensorflow_gpu-1.8.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.10.0	7	9
tensorflow_gpu-1.7.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.9.0	7	9
tensorflow_gpu-1.6.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.9.0	7	9
tensorflow_gpu-1.5.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.8.0	7	9
tensorflow_gpu-1.4.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.5.4	6	8
tensorflow_gpu-1.3.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.4.5	6	8
tensorflow_gpu-1.2.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.4.5	5.1	8
tensorflow_gpu-1.1.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.4.2	5.1	8
tensorflow_gpu-1.0.0	2.7, 3.3-3.6	GCC 4.8	Bazel 0.4.2	5.1	8

- **适用于单系统多用户的安装方式：基于VirtualEnv的安装-日常启动与关闭**

假定设置的virtualenv环境的目录为~/tensorflow，启动方法：
$ cd ~/tensorflow
$ source bin/activate
终止方法：
(tensorflow)$ deactivate # 停用 virtualenv

- **简易实例**

在进入Python交互界面之后，先通过import操作加载TensorFlow:
import tensorflow as tf
小例子:

```
>>> a = tf.constant([1.0,2.0], name="a")
>>> b = tf.constant([2.0,3.0], name="b")
>>> result = a + b
>>> sess = tf.Session()
>>> sess.run(result)
array([ 3.,  5.], dtype=float32)
```

要输出得到结果，不能简单地直接输出result,而需要先生成一个会话(session),
并通过一个这个会话(session)来计算结果。
到此，就实现了一个非常简单的TensorFlow 模型。

Part 2 开发（Python）

- **计算图**

  TensorFlow 程序通常被组织成一个构建阶段和一个执行阶段。在构建阶段, op（操作） 的执行步骤被描述成一个图。 在执行阶段, 使用会话执行执行图中的 op。

- **计算图-构建阶段**

  构建图的第一步, 是创建源 op (source op)。TensorFlow Python 库有一个默认图 (default graph), op 构造器可以为其增加节点。 这个默认图对 许多程序来说已经足够用了。

```
import tensorflow as tf
# 构造器的返回值代表该常量 op 的返回值。
matrix1 = tf.constant([[3., 3.]])
# 创建另外一个常量 op, 产生一个 2x1 矩阵。
matrix2 = tf.constant([[2.],[2.]])
# 创建一个矩阵乘法 matmul op , 把 ‘matrix1’ 和 ‘matrix2’ 作为输入。 返回值 ‘product’ 代表矩阵乘法的结果。
product = tf.matmul(matrix1, matrix2)
```

  默认图现在有三个节点, 两个 constant() op, 和一个matmul() op。 为了真正进行矩阵相乘运算, 并得到矩阵乘法的结果, 必须在会话里启动这个图.

- **计算图-执行阶段**

构造阶段完成后, 才能启动图。 启动图的第一步是创建一个 Session 对象, 如果无任何创建参数, 会话构造器将启动默认图。

```
# 启动默认图。
sess = tf.Session()
# 调用 sess 的 ‘run()’ 方法来执行矩阵乘法 op, 传入 ‘product’ 作为该方法的参数。‘product’ 代表了矩阵乘法 op 的输出, 传入它是向方法表明, 我们希望取回矩阵乘法 op 的输出。函数调用 ‘run(product)’ 触发了图中三个 op (两个常量 op 和一个矩阵乘法 op) 的执行。result = sess.run(product)
print result
# 任务完成, 关闭会话.
sess.close()
```

Session 对象在使用完后需要关闭以释放资源. 除了显式调用 close 外, 也可以使用 "with" 代码块来自动完成关闭动作.

```
with tf.Session() as sess:
        result = sess.run([product])
        print result
```

- **Tensor**

TensorFlow 程序使用 tensor 数据结构来代表所有的数据, 计算图中, 操作间传递的数据都是 tensor。可以把 TensorFlow tensor 看作是一个 n 维的数组或列表. 一个 tensor 由阶形状、类型组成。

- **变量:创建、初始化**

  当创建一个变量时，将一个张量作为初始值传入构造函数Variable()。
  weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),name="weights")初始化
  变量的初始化是：tf.initialize_all_variables()。它必须在模型的其它操作运行之前先明确地完成。最简单的方法就是添加一个给所有变量初始化的操作，并在使用模型之前首先运行那个操作。 在完全构建好模型并加载之后再运行那个操作。

```
# Create two variables.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
...
# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()

# Later, when launching the model
with tf.Session() as sess:
  # Run the init operation.
  sess.run(init_op)
  ...
  # Use the model
  ...
```

- **变量:保存**

最简单的保存和恢复模型变量的方法是使用tf.train.Saver对象。
模型的变量存储在二进制文件里，主要包含从变量名到tensor值的映射关系。
当你创建一个Saver对象时，你可以选择性地为二进制文件中的变量挑选变量名。默认情况下，变量名是每个变量name属性的值。
用tf.train.Saver()创建一个Saver来管理模型中的所有变量。

```
# Create some variables。
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()
# Add ops to save and restore all the variables.
saver = tf.train.Saver()
# Later, launch the model, initialize the variables, do some work, save the variables to disk.
with tf.Session() as sess:
        sess.run(init_op)
        # Do some work with the model.
        ..
        # Save the variables to disk.
        save_path = saver.save(sess, "/tmp/model.ckpt") 修改为当前用户下子目录
        print "Model saved in file: ", save_path
```

- **变量:恢复**

用同一个Saver对象来恢复变量。注意，当你从文件中恢复变量时，不需要事先对它们做初始化。

```
# Create some variables.
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, use the saver to restore variables from disk, and
# do some work with the model.with tf.Session() as sess:
# Restore variables from disk.
saver.restore(sess, "/tmp/model.ckpt") 修改为当前用户下子目录
print "Model restored."
# Do some work with the model
...
```

Part 3 可视化（Python）

- **图像化**

  TensorBoard 涉及到的运算，通常是在训练庞大的深度神经网络中出现的复杂而又难以理解的运算。为了更方便 TensorFlow 程序的理解、调试与优化，发布了一套叫做 TensorBoard 的可视化工具。可以用 TensorBoard 来展现 网络架构，绘制生成的定量指标图等。

- **数据序列化**

  TensorBoard 通过读取将数据序列化后的文件来运行，此文件包含了在 TensorFlow 运行中涉及到的主要数据。下面是 TensorBoard 将数据序列化的大体生命周期。

  首先，创建网络架构计算图，然后再选择节点进行数据附加操作。比如，假设正在训练一个卷积神经网络，用于识别 Mnist 标签，当希望显示学习率的变化时，可以通过向相关节点附加scalar_summary操作来记录学习率的数值；当希望显示一个卷积层中梯度权重的分布时，可以通过附加 histogram_summary操作来收集梯度信息。

  然后，合并操作。在TensorFlow中，所有的操作只有当执行，或者另一个操作依赖于它的输出时才会运行。为了生成汇总信息，需要运行所有这些节点，可以使用tf.merge_all_summaries来将他们合并为一个操作。

  最后，执行合并命令。它会依据特定步骤将所有数据生成一个序列化的对象。最后，为了将汇总数据写入磁盘，需要将汇总的对象传递给tf.train.Summarywriter。

Part 3 可视化（Python）

- **数据序列化-实例**

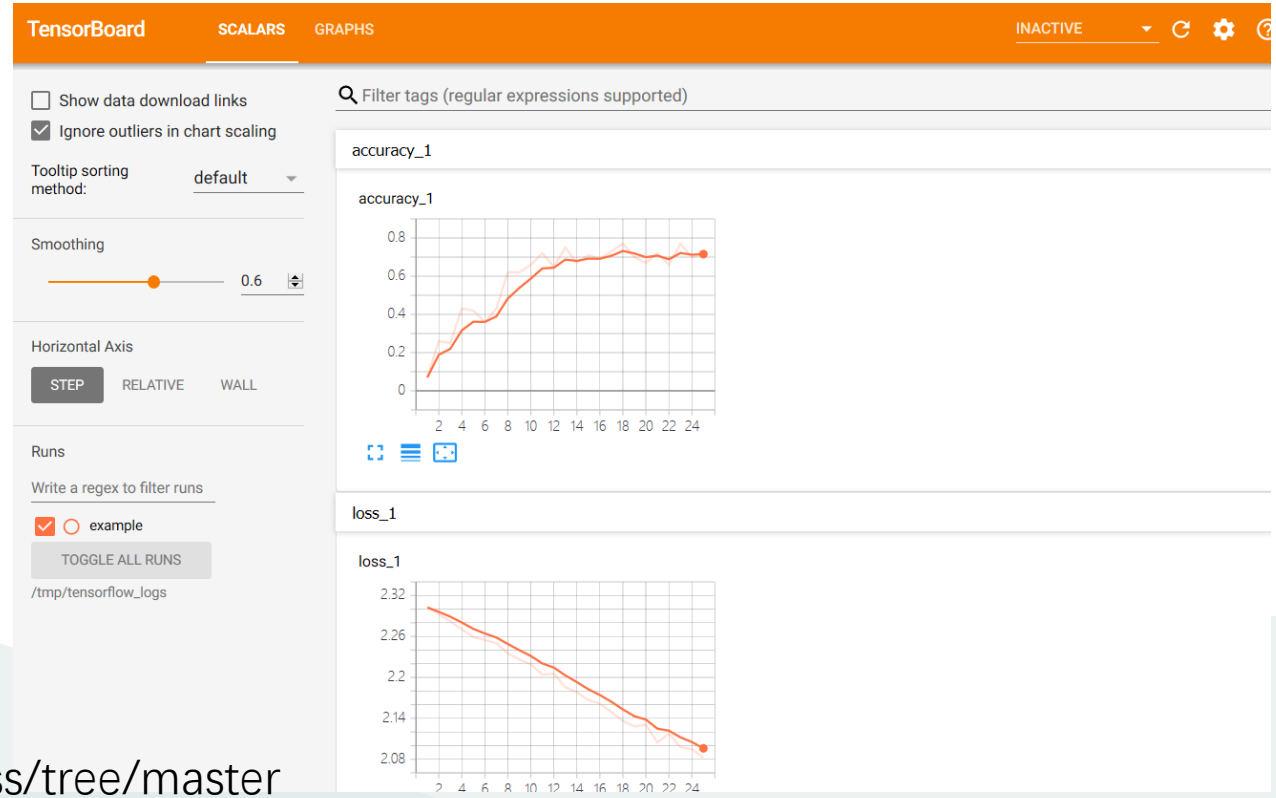
```
# Create a summary to monitor cost tensor
tf.summary.scalar("loss", cost)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("accuracy", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop), cost op (to get loss value)
        # and summary nodes
        _, c, summary = sess.run([optimizer, cost, merged_summary_op],
                                 feed_dict={x: batch_xs, y: batch_ys})
        # Write logs at every iteration
        summary_writer.add_summary(summary, epoch+ 1)
```

设置保存路径为当前用户下子目录

Part 3 可视化（Python）

- **启动TensorBoard**

输入下面的指令来启动TensorBoard
tensorboard --logdir=path/to/log-directory
这里的参数 logdir 指向 SummaryWriter 序列化数据的存储路径。如果logdir目录的子目录中包含另一次运行时的数据，那么 TensorBoard 会展示所有运行的数据。
一旦 TensorBoard 开始运行，你可以通过在浏览器中输入 localhost:6006 来查看 TensorBoard。
进入 TensorBoard 的界面时，你会在上方看到导航选项卡，每一个选项卡将展现一组可视化的序列化数据集 。
在本地浏览器输入10.108.14.201:6006 即可查看数据的可视化信息。



代码示例https://github.com/HYWZ36/tensorboard_loss/tree/master

Part 4 GPU（Python）

- **使用GPU（Python）**

GPU在深度学习中可处理大数据量的复杂运算。在脚本中输入下述命令可指定某一型号GPU参与到程序的运算中去：

```
import os
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ['CUDA_VISIBLE_DEVICES'] = "0“
```

服务器有4个GPU，序号为0,1,2,3， os.environ[‘CUDA_VISIBLE_DEVICES‘] 填入一个GPU序号即可，填写多个会浪费计算资源。

Part 5 共享变量

通常需要共享大量变量集并且如果还想在同一个地方初始化这所有的变量，该怎么做呢？
可以使用tf.variable_scope() 和tf.get_variable()两个方法来实现这一点。

- **变量作用域机制**

变量作用域机制在TensorFlow中主要由两部分组成：
tf.get_variable(<name>, <shape>, <initializer>)：通过所给的名字创建或是返回一个变量，而不是直接调用tf.Variable 。
定义conv_relu函数：

```
def conv_relu(input, kernel_shape, bias_shape):
    # Create variable named "weights".
    weights = tf.get_variable("weights", kernel_shape,
        initializer=tf.random_normal_initializer())
    # Create variable named "biases".
    biases = tf.get_variable("biases", bias_shape,
        initializer=tf.constant_intializer(0.0))
    conv = tf.nn.conv2d(input, weights,
        strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)
```

Part 5 共享变量

- **变量作用域机制**

  tf.variable_scope(<scope_name>)：为变量名指定命名空间。

```
def my_image_filter(input_images):
    with tf.variable_scope("conv1"):
        # Variables created here will be named "conv1/weights", "conv1/biases".
        relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
    with tf.variable_scope("conv2"):
        # Variables created here will be named "conv2/weights", "conv2/biases".
        return conv_relu(relu1, [5, 5, 32, 32], [32])
```

当调用 my_image_filter() 时共享参数出错。

```
result1 = my_image_filter(image1)
result2 = my_image_filter(image2)
# Raises ValueError(... conv1/weights already exists ...)
```

tf.get_variable()会检测已经存在的变量是否已经共享。如果想共享它们，需要通过reuse_variables()这个方法来指定。

tf.get_variable_scope().reuse_variables()

```
with tf.variable_scope("image_filters") as scope:
    result1 = my_image_filter(image1)
    scope.reuse_variables()
    result2 = my_image_filter(image2)
```