目录

第一章 绪论

1.1:数据挖掘简介

人类正被数据淹没，却饥渴于知识。随着数据库技术的应用越来越普及，人们正逐步陷入“数据丰富，知识贫乏”的尴尬境地。知识信息的“爆炸”给人类带来莫大益处，但也带来不少弊端，造成知识信息的“污染”。面临浩瀚无际而被污染的数据，人们呼唤从数据汪洋中来一个去粗取精、去伪存真的技术。在这种形势下，数据挖掘应运而生。数据挖掘就是指从大量的、不完全的、有噪声的、模糊的、随机的实际应用数据中，提取隐含在其中的，人们事先不知道的，但又是潜在有用的，目标明确、 针对性强、精炼准确的信息和知识的过程。 数据与知识贫乏导致了知识发现和数据挖掘的出现，当人们进入 21 世纪以后，可以预计知识发现与数据挖掘的研究又将形成一个新的高潮。

数据挖掘是一门多学科交叉的领域。一方面，数据挖掘以计算技术的发展为首要条件，没有数据的有效组织，从一堆数据垃圾中发现有用的知识是痴人说梦；没有大量计算算法的支持，即使是简单的查询也会耗时巨大，更不用说发现有用的模式。另一方面，即使数据得到有效的组织，计算算法足够先进，要想发现数据中隐藏的有用信息，还必须综合利用统计学、模式识别、人工智能、机器学习、神经网络等学科的专业知识。比如数据挖掘使用的分析方法，有相当大比重是靠统计学中的多元分析来支撑的，由统计理论衍生出来的。当然，所有这些学科的发展必然会从不同的角度关 注数据分析技术的进展，数据挖掘也为这些学科的发展提供了新的机遇和挑战。

数据挖掘是用于数据处理的一种新的思维方法和技术手段，它是在现实生活中各种数据量不断增长，以及以数据库技术为核心的信息技术逐渐成熟的背景下产生的。数据挖掘可以帮助用户发现隐藏在数据库中的规律和模式，它融合了人工智能、统计、机器学习、模式识别和数据库等多种学科的理论、方法与技术，通过对数据的归纳、分析和推理，从中发掘出潜在的模式，帮助决策者调整策略，进行正确的决策。

但是，我们也必须看到，要研究数据挖掘，必须强调所用方法的概念和属性，而不是机械的应用不同的数据挖掘工具。对方法、模型以及它们怎样运转及运转原理的 深入理解是有效和成功运用数据挖掘技术的先决条件。任何在数据挖掘领域的研究者和实践者都要意识到这些问题，以便成功地应用一种特定的方法，理解一种方法的 局限性，或者开发新技术。

1.2 数据挖掘所用到的技术

想要实现数据挖掘，就需要用到许多技术比如：统计学、机器学习等。

1.2.1 统计学

统计学研究数据的收集、分析、解释和表示。其实，数据挖掘与统计学具有天然联系。统计模型是一组数学函数，它们用随机变量及其概率分布刻画目标类对象的行为。统计模型广泛用于对数据建模。例如，在像数据特征化和分类这样的数据挖掘任务中，可以建立目标类的统计模型。换言之，这种统计模型可以是数据挖掘任务的结果。反过来，数据挖掘任务也可以建立在统计模型之上。例如，我们可以使用统计模型对噪声和缺失的数据值建模。于是，在大数据集中挖掘模式时，数据挖掘过程可以使用该模型来帮助识别数据中的噪声和缺失值。

统计学研究开发一些使用数据和统计模型进行预测和预报的工具。统计学方法可以用来汇总或描述数据集。对于从数据中挖掘各种模式，以及理解产生和影响这些模式的潜在机制，统计学是有用的。推理统计学用某种方式对数据建模，解释观测中的随机性和确定性，并用来提取关于所考察的过程或总体的结论。

统计学方法也可以用来验证数据挖掘结果。例如分类或者预测模型之后，应该使用统计假设检验来验证模型。

在数据挖掘中使用统计学方法并不简单。通常，一个巨大的挑战是如何把统计学方法用于大型数据集。许多统计学方法都具有很高的计算复杂度。当这些方法应用于分布在多个逻辑或物理站点熵的大型数据集时，应该小心地设计和调整算法，以降低计算开销。

1.2.2 机器学习

某些方面上来说，数据挖掘与机器学习在之间存在着许多相同之处。在这里我们介绍一些与数据挖掘高度相关的，经典的机器学习问题。

- 监督学习：即在机器学习模型的训练过程中所使用的训练集是有标签，即标准答案的。也可以理解成告诉哪些数据的答案是 A，哪些数据的答案是 B，然后让机器学习程序来自己学习其中的规律。
- 无监督学习：本质上是聚类的同义词。学习过程是无监督的，因为输入样本是没有标签的。典型地，我们可以使用聚类发现数据中的类。例如一个无监督学习方法可以取一个手写数字图像集合作为输入。假设它找出了 10 个数据簇，这些簇可以分别对应 0-9 这 10 个不同的数字。然而，由于训练数据并无标记，因此学习到的模型并不能告诉我们发现的簇的语义。
- 半监督学习：是一类机器学习技术，在学习模型时，它使用标记和未标记的样本。其中标记的样本用来学习模型，未标记的样本用来进一步改进模型。

你可能已经看出，数据挖掘与机器学习有许多相似之处。对于分类和聚类任务，机器学习研究通常关注模型的准确率。除准确率之外，数据挖掘研究非常强调挖掘方法在大型数据集上的有效性，以及处理复杂数据类型的方法。

1.2.3 信息检索

信息检索是搜索文档或文档中信息的科学。文档可以是文本或多媒体，并且可能驻留在 Web 上。传统的信息检索与数据库系统之间的差别有两点：信息检索假定所搜索的数据是无结构的；信息检索查询主要用关键词，没有复杂的结构。

信息检索的典型方法是使用概率模型。例如，文本文档可以看作词的包，即出现在文档中的词的多重集合。文档的语言模型是生成文档中词的包的概率密度函数。两个文档之间的相似度可以用对应的语言模型之间的相似性度量。

此外，一个文本文档集的主题可以用词汇表上的概率分布建模，即主题模型。一个文本文档可以涉及多个主题，可以看作多个主题混合的模型。通过集成信息检索模型和数据挖掘技术，我们可以找出文档集中的主要问题，对集合中的每个文档，找出所涉及的主要主题。

1.3 数据挖掘的应用场景

数据挖掘技术可以为决策、过程控制、信息管理和查询处理等任务提供服务，一个有趣的应用范例是“尿布与啤酒”的故事。为了分析哪些商品顾客最有可能一起购买，一家名叫 WalMart 的公司利用自动数据挖掘工具，对数据库中的大量数据进行分析后，意外发现，跟尿布一起购买最多的商品竟是啤酒。为什么两件风马牛不相及的商品会被人一起购买？原来，太太们常叮嘱她们的丈夫，下班后为小孩买尿布，而丈夫们在买尿布后又随手带回了两瓶啤酒。既然尿布与啤酒一起购买的机会最多，商店就将它们摆放在一起，结果，尿布与啤酒的销售量双双增长。这里，数字挖掘技术功不可没。一般来说，数据挖掘的应用场景有电子政务、零售业、网站等，具体如下。

1.3.1 电子政务

建立电子化政府，推动电子政务的发展，是电子信息技术应用到政府管理的必然趋势。实践经验表明，政府部门的决策越来越依赖于对数据的科学分析。发展电子政务，建立决策支持系统，利用电子政务综合数据库中存储的大量数据，通过建立正确的决策体系和决策支持模型，可以为各级政府的决策提供科学的依据，从而提高各项政策制定的科学性和合理性，以达到提高政府办公效率、促进经济发展的目的。为此，在政府决策支持方面，需要不断吸纳新的信息处理技术，而数据挖掘正是实现政府决策支持的核心技术。以数据挖掘为依托的政府决策支持系统，将发挥重要的作用。

电子政务位于世界各国积极倡导的“信息高速公路”五个领域（电子政务、电子商务、远程教育、远程医疗、电子娱乐）之首，说明政府信息化是社会信息化的基础。电子政务包括政府的信息服务、电子贸易、电子化政府、政府部门重构、群众参与政府五个方面的内容。将网络数据挖掘技术引入电子政务中，可以大大提高政府信息化水平，促进整个社会的信息化。具体体现在以下几个方面：

- 政府的电子贸易 在服务器以及浏览器端日志记录的数据中隐藏着模式信息，运用网络用法挖掘技术可以自动发现系统的访问模式和用户的行为模式，从而进行预测分析。例如，通过评价用户对某一信息资源浏览所花费的时间，可以判断出用户对何种资源感兴趣;对日志文件所收集到的域名数据，根据国家或类型进行分类分析;应用聚类分析来识别用户的访问动机和访问趋势等。这项技术已经有效地运用在政府电子贸易中。
- 网站设计 通过对网站内容的挖掘，主要是对文本内容的挖掘，可以有效地组织网站信息，如采用自动归类技术实现网站信息的层次性组织;同时可以结合对用户访问日志记录信息的挖掘，把握用户的兴趣，从而有助于开展网站信息推送服务以及个人信息的定制服务，吸引更多的用户。
- 搜索引擎 网络数据挖掘是目前网络信息检索发展的一个关键。如通过对网页内容挖掘，可以实现对网页的聚类、分类，实现网络信息的分类浏览与检索；同时，通过对用户所使用的提问式的历史记录的分析，可以有效地进行提问扩展，提高用户的检索效果；另外，运用网络内容挖掘技术改进关键词加权算法，提高网络信息的标引准确度，从而改善检索效果。
- 决策支持 为政府重大政策出台提供决策支持。如，通过对网络各种经济资源的挖掘，确定未来经济的走势，从而制定出相应的宏观经济调控政策。

1.3.2 零售业

通过条形码、编码系统、销售管理系统、客户资料管理及其它业务数据中，可以收集到关于商品销售、客户信息、货存单位及店铺信息等的信息资料。数据从各种应用系统中采集，经条件分类，放到数据仓库里，允许高级管理人员、分析人员、采购人员、市场人员和广告客户访问，利用DM工具对这些数据进行分析，为他们提供高效的科学决策工具。如对商品进行购物篮分析，分析哪些商品是顾客最有希望一起购买的。如被业界和商界传诵的经典----Wal-Mart的“啤酒和尿布”，就是数据挖掘透过数据找出人与物间规律的典型。在零售业应用领域，利用DW、DM会在很多方面有卓越表现：

- 了解销售全局：通过分类信息——按商品种类、销售数量、商店地点、价格和日期等了解每天的运营和财政情况，对销售的每一点增长、库存的变化以及通过促销而提高的销售额都可了如指掌。零售商店在销售商品时，随时检查商品结构是否合理十分重要，如每类商品的经营比例是否大体相当。调整商品结构时需考虑季节变化导致的需求变化、同行竞争对手的商品结构调整等因素。
- 商品分组布局：分析顾客的购买习惯，考虑购买者在商店里所穿行的路线、购买时间和地点、掌握不同商品一起购买的概率；通过对商品销售品种的活跃性分析和关联性分析，用主成分分析方法，建立商品设置的最佳结构和商品的最佳布局。
- 降低库存成本：通过数据挖掘系统，将销售数据和库存数据集中起来，通过数据分析，以决定对各个商品各色货物进行增减，确保正确的库存。数据仓库系统还可以将库存信息和商品销售预测信息，通过电子数据交换（EDI）直接送到供应商那里，这样省去商业中介，而且由供应商负责定期补充库存，零售商可减少自身负担。
- 市场和趋势分析：利用数据挖掘工具和统计模型对数据仓库的数据仔细研究，以分析顾客的购买习惯、广告成功率和其它战略性信息。利用数据仓库通过检索数据库中近年来的销售数据，作分析和数据挖掘，可预测出季节性、月销售量。

有效的商品促销：可以通过对一种厂家商品在各连锁店的市场共享分析，客户统计以及历史状况的分析，来确定销售和广告业务的有效性。通过对顾客购买偏好的分析，确定商品促销的目标客户，以此来设计各种商品促销的方案，并通过商品购买关联分析的结果，采用交叉销售和向上销售的方法，挖掘客户的购买力，实现准确的商品促销。

1.3.3 网站

随着Web技术的发展，各类电子商务网站风起云涌。建立一个电子商务网站并不困难，困难的是如何让您的电子商务网站有效益。要想有效益就必须吸引客户，增加能带来效益的客户忠诚度。电子商务业务的竞争比传统的业务竞争更加激烈，原因有很多方面，其中一个因素是客户从一个电子商务网站转换到竞争对手那边，只需要点击几下鼠标即可。网站的内容和层次、用词、标题、奖励方案、服务等任何一个地方都有可能成为吸引客户、同时也可能成为失去客户的因素。而同时电子商务网站每天都可能有上百万次的在线交易，生成大量的记录文件（Log files）和登记表，如何对这些数据进行分析和挖掘，充分了解客户的喜好、购买模式，甚至是客户一时的冲动，设计出满足不同客户群体需要的个性化网站，进而增加其竞争力，几乎变得势在必行。若想在竞争中生存进而获胜，就要比您的竞争对手更了解客户。

在对网站进行数据挖掘时，所需要的数据主要来自于两个方面：一方面是客户的背景信息，此部分信息主要来自于客户的登记表；而另外一部分数据主要来自浏览者的点击流（Click-stream），此部分数据主要用于考察客户的行为表现。但有的时候，客户对自己的背景信息十分珍重，不肯把这部分信息填写在登记表上，这就会给数据分析和挖掘带来不便。在这种情况之下，就不得不从浏览者的表现数据中来推测客户的背景信息，进而再加以利用。

第二章 认识数据

2.1 数据与属性

一般而言，一个数据集一般由多条数据组成，而一条数据也一般由多个属性构成。如果将一个数据集看成是一个表格，那么表格中的每一行表示一条数据，而表格中的每一列表示数据的一个属性。如下图所示的泰坦尼克数据集中有 5 条数数据，数据有 12 个属性。当然，我们通常将属性称为特征。

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

根据特征的特点可以将其划分成**categorical**特征、**ordinal**特征、**numeric**特征。

categorical特征

categorical特征，顾名思义即类别特征。注意：这里的类别指的是没有顺序的类别。如泰坦尼克数据集中的 `Survived` 、 `Sex` 、 `Cabin` 、 `Embarked` 都是categorical特征。因为这些特征的值都代表某种类别。
如 `Sex` 中的 `male` 和 `female` 分别代表男性和女性，而且男性与女性之间没有顺序关系。

ordinal特征

ordinal特征和categorical特征一样都代表类别特征，只不过ordinal特征表示的是具有顺序属性的类别特征。如泰坦尼克数据集中的 `Pclass` 、 `SibSp` 、 `Parch` 属于ordinal特征。例如 `Pclass` 表示乘客的船舱的等级， `3、2、1` 分别表示三等舱、二等舱和一等舱。很明显，一等舱的高级程度是高于其他两种舱的，所以是 `Pclass` 是ordinal特征。

numeric特征

numeric特征表示的数值型特征，这就很好理解了。只要该特征的值是数值型的，则该特征为numeric特征。如 `Age` 、 `Fare` 都是numeric特征。

2.2 数据的基本统计指标

在进行数据挖掘之前，通常需要先了解数据集中数据的分布。所谓的分布，就是查看数据集中特征的一些统计指标。常见的统计指标有均值，中值，标准差，方差等。

假设现在有这样的一份长沙房价数据，并接下来使用这份数据来讲解什么是均值、中值、标准差和方差。

编号	地区	建筑面积	总价
1	开福区	120	900000
2	岳麓区	111	700000
3	天心区	93	600000
4	雨花区	140	1200000
5	开服区	121	910000
6	岳麓区	87	500000

均值

均值即数据表格中的某一列所有的值相加再除以数据条数。反映的是某个特征的特征值的平均水平。如表格中总价的均值为: `(900000+700000+600000+1200000+910000+500000)/6=801666.7` 。也就是说长沙的平均房价为 80 万左右。

中值

中值即对数据表格中某一列所有的值进行排序后，排在中间位置的值。反映的是某个特征的特征值的中等水平。如表格中建筑面积经过排序后为 `87, 93, 111, 120, 121, 140` ，那么建筑面积的中值就是 111 。也就是说整个数据集给出的信息是，长沙中等水平的房子的面积为 111 平。

方差

方差表示的是表格中某一列所有的值的分散程度，方差越大说明越分散。方差的计算公式如下(其中μ表示均值)：

$$\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2$$

如表格中总价的方差为: `((900000-801666.7)^2+(700000-80166.7)^2+(600000-80166.7)^2+(1200000-80166.7)^2+(910000-80166.7)^2+(500000-80166.7)^2)/6=574242757455.5568` 。从方差的值来看，数据中体现了长沙的房价的分散程度比较大，并没有集中在均价的水平。

标准差

标准差即方差的算术平方根。如表格中总价的标准差为: `757788.0689583049` 。同样，标准差越大说明数据越分散。

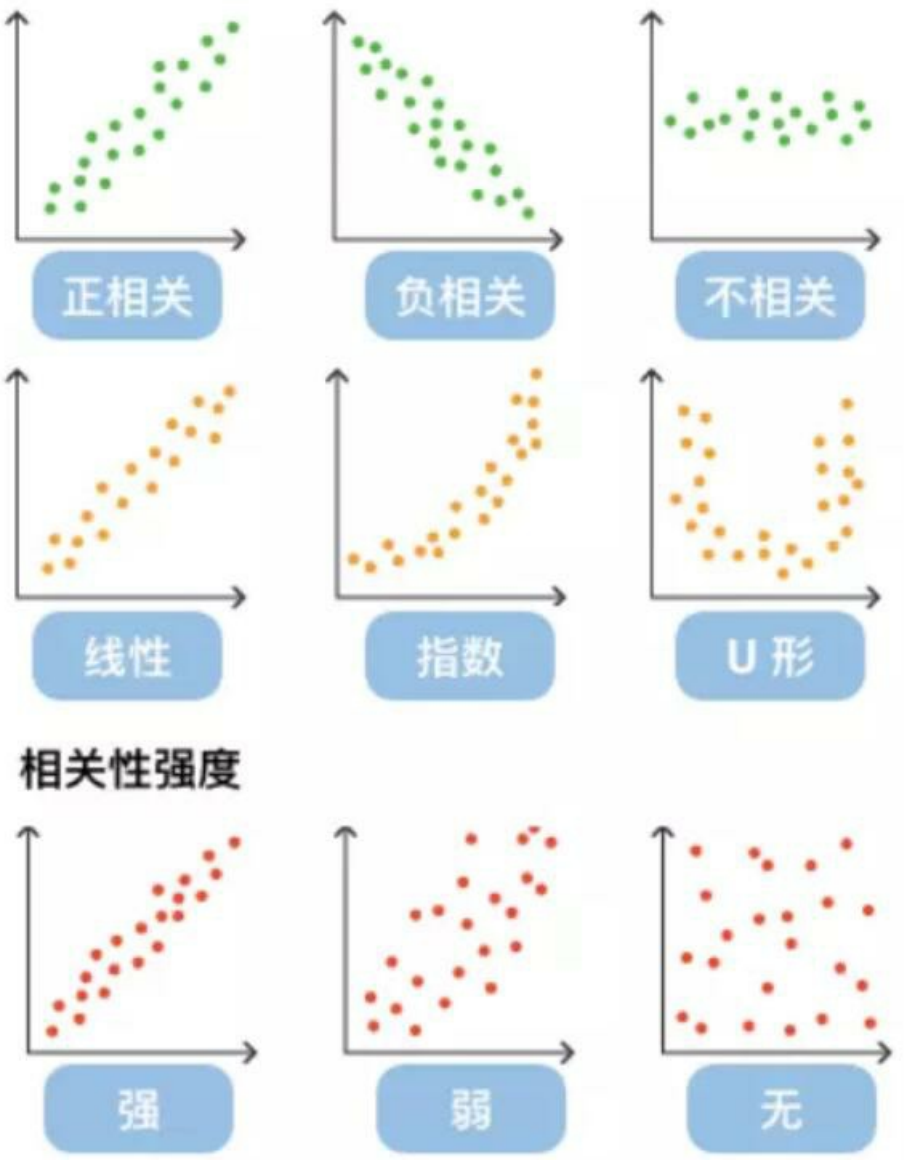
2.3 数据可视化

人类都是视觉动物，虽然常见的统计量能够反映出一些数据的特点，但是并不直观，如果能够将数据中的特征的分布或者特征与特征之间的关系可视化出来，那么对于理解数据来说，算是比较舒服的一件事情了。所以接下来就介绍一些常见的可视化图形，都表示了什么样的信息。

散点图

散点图也叫 x-y 图，它将所有的数据以点的形式展现在直角坐标系上，以显示变量之间的相互影响程度，点的位置由变量的数值决定。

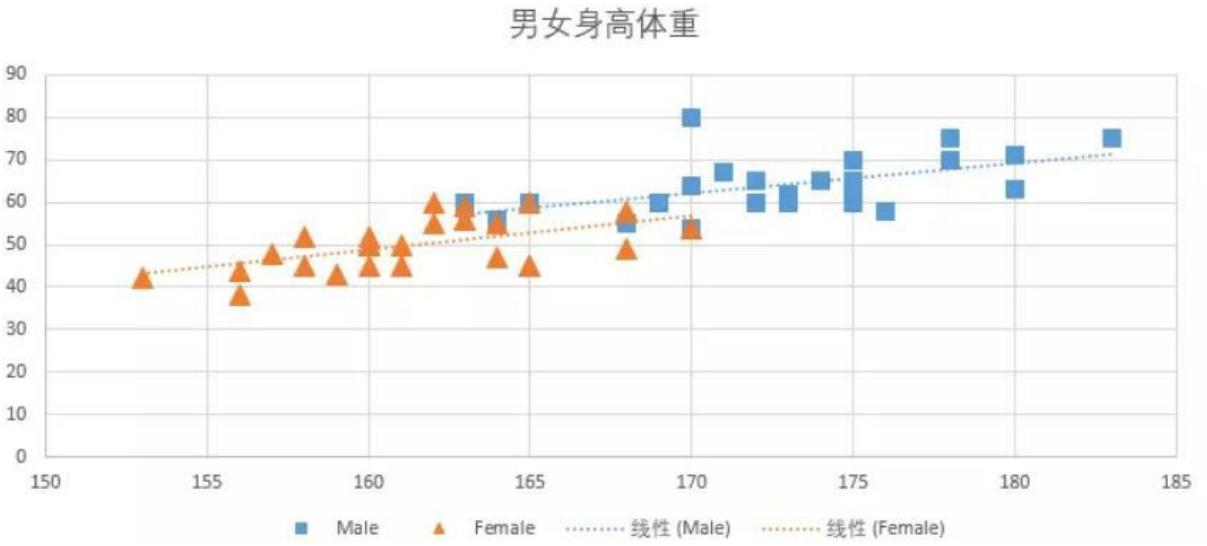
通过观察散点图上数据点的分布情况，我们可以推断出变量间的相关性。如果变量之间不存在相互关系，那么在散点图上就会表现为随机分布的离散的点，如果存在某种相关性，那么大部分的数据点就会相对密集并以某种趋势呈现。数据的相关关系主要分为：正相关（两个变量值同时增长）、负相关（一个变量值增加另一个变量值下降）、不相关、线性相关、指数相关等，表现在散点图上的大致分布如下图所示。那些离点集群较远的点我们称为离群点或者异常点。



下面来看一个例子，假设现在有这样一份数据。其中包括性别，年龄，身高和体重。

Gender	Age	Height(cm)	Weight(kg)
Male	21	163	60
Male	22	164	56
Male	21	165	60
Male	23	168	55
Male	21	169	60
Male	21	170	54
Male	23	170	80
Male	23	170	64
Male	22	171	67
Male	22	172	65
Male	23	172	60
Male	21	172	60
Male	23	173	60
Male	22	173	62
Male	21	174	65
Male	22	175	70
Male	22	175	70
Male	22	175	65
Male	23	175	60
Male	21	175	62
Male	21	176	58
Male	21	178	70

如果我们使用身高和体重这两个特征来画出散点图，则会有如下可视化结果：

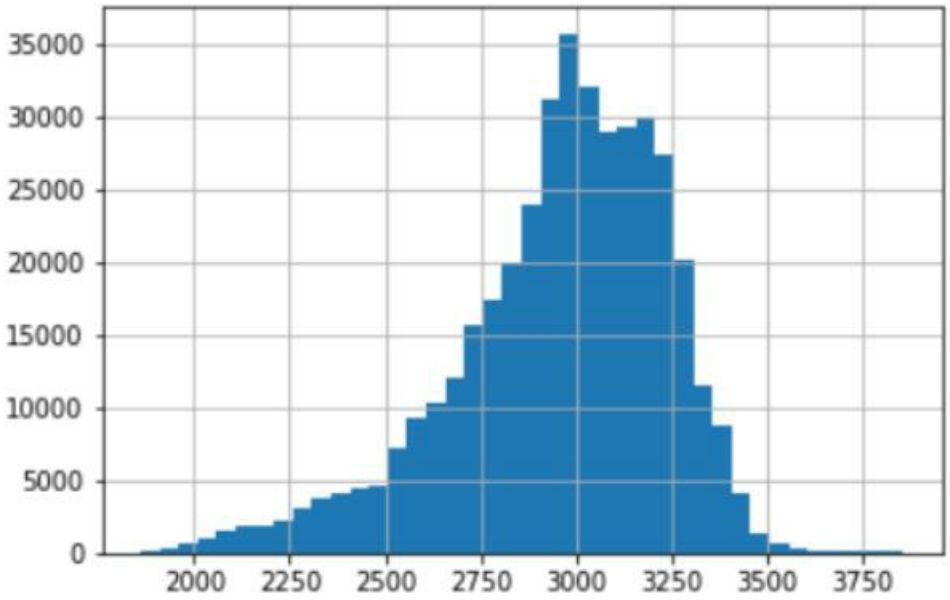


从可视化结果可以看出，不管是男生还是女生仿佛都遵循这一个规律，就是身高越高体重就越重。而且细心一点，可以看出身高 170 体重为 90 的人应该是属于肥胖了。

直方图

直方图可以看成是描述数据分布的分布图，横轴代表特征的数值，纵轴代表该特征值在数据集中出现的频率或者次数。

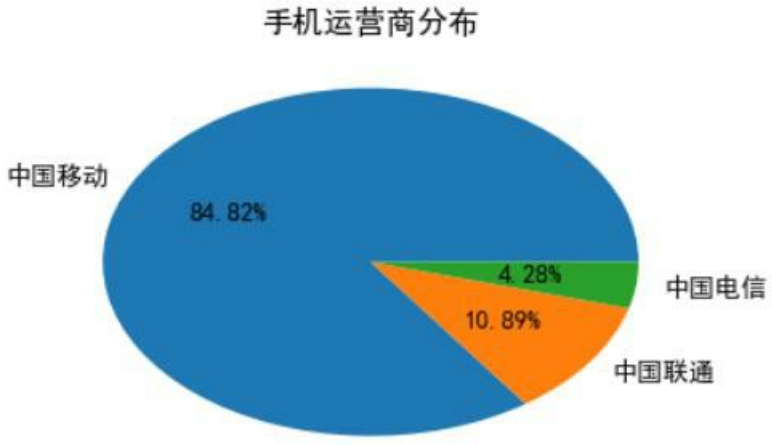
例如这是罗斯福国家公园树木数据集中树木所处地的海拔高度的直方图。



从图中可以看出，大部分树木都在海拔 3000 米左右的地方生长。而且整个海拔的分布近似于正态分布。这也是符合常理的，因为只有极少部分品种的树木需要在非常低或者非常高的海拔的条件下才能生长。

饼图

当想要看看某个categorical特征的值的占比时，可以尝试使用饼图来进行可视化。例如以下饼图为某县手机运营商的市场份额分布图。



从图中可以看出，该县的手机运营商的市场基本上已经被中国移动给霸占了。说明当地的中国移动可能更会做宣传，或者服务质量更好。

第三章 数据预处理

3.1 为什么要数据预处理

数据挖掘其实就是从数据中学习到规律，再将学习到的规律对未知的数据进行分析。数据的质量直接影响到模型学习的好坏，而我们最开始获取的数据其中绝大多数是“有毛病”的，不利于后期进行分析。所以我们在分析前需要进行数据的预处理。

`sklearn.preprocessing` 包提供了几个常用的函数和转换类型，用它们将一个原始的特征向量转化为一个更适于数据分析的表示形式。一般来说，学习算法收益于数据集的标准形式。如果数据中存在异常点，稳健的数据规范或转换是更适合的。

3.2 标准化

为什么要进行标准化

对于大多数数据挖掘算法来说，数据集的标准化是基本要求。这是因为，如果特征不服从或者近似服从标准正态分布（即，零均值、单位标准差的正态分布）的话，算法的表现会大打折扣。实际上，我们经常忽略数据的分布形状，而仅仅做零均值、单位标准差的处理。在一个机器学习算法的目标函数里的很多元素所有特征都近似零均值，方差具有相同的阶。如果某个特征的方差的数量级大于其它的特征，那么，这个特征可能在目标函数中占主导地位，这使得模型不能从其它特征有效地学习。

Z-score标准化

这种方法基于原始数据的均值 `mean` 和标准差 `standard deviation` 进行数据的标准化。对特征 `A` 的原始值 `x` 使用 `z-score` 标准化，将其值标准化为到 `x'` 。 `z-score` 标准化方法适用于特征 `A` 的最大值和最小值未知的情况，或有超出取值范围的离群数据的情况。将数据按其特征(按列进行)减去其均值，然后除以其方差。最后得到的结果是，对每个特征/每列来说所有数据都聚集在 `0` 附近，方差值为 `1` 。数学公式如下：

$$x' = \frac{x - x_{mean}}{x_{std}}$$

函数 `scale` 为数组形状的数据集的标准化提供了一个快捷实现：

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])
X_scaled = preprocessing.scale(X_train)

>>>X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

经过缩放后的数据具有零均值以及标准方差:

```
>>> X_scaled.mean(axis=0)
array([ 0.,  0.,  0.])

>>> X_scaled.std(axis=0)
array([ 1.,  1.,  1.])
```

Min-max标准化

`Min-max` 标准化方法是对原始数据进行线性变换。设 `minA` 和 `maxA` 分别为特征 `A` 的最小值和最大值，将 `A` 的一个原始值 `x` 通过 `min-max` 标准化映射成在区间 `[0,1]` 中的值 `x'` ，其公式为：

$$x' = \frac{x - x_{min}}{x_{max} - xmin}$$

可以使用 `MinMaxScaler` 实现，以下是一个将简单的数据矩阵缩放到 `[0, 1]` 的例子:

```
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

min_max_scaler = preprocessing.MinMaxScaler()
X_train_minmax = min_max_scaler.fit_transform(X_train)

>>> X_train_minmax
array([[ 0.5       ,  0.        ,  1.        ],
       [ 1.        ,  0.5       ,  0.33333333],
       [ 0.        ,  1.        ,  0.        ]])
```

MaxAbs标准化

`MaxAbs` 的工作原理与 `Min-max` 非常相似，但是它只通过除以每个特征的最大值将训练数据特征缩放至 `[-1, 1]` 范围内，这就意味着，训练数据应该是已经零中心化或者是稀疏数据。公式如下：

$$x' = \frac{x}{x_{max}}$$

以下是使用上例中数据，并对其进行 `MaxAbs` 标准化的例子:

```
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

max_abs_scaler = preprocessing.MaxAbsScaler()
X_train_maxabs = max_abs_scaler.fit_transform(X_train)

>>> X_train_maxabs
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
```

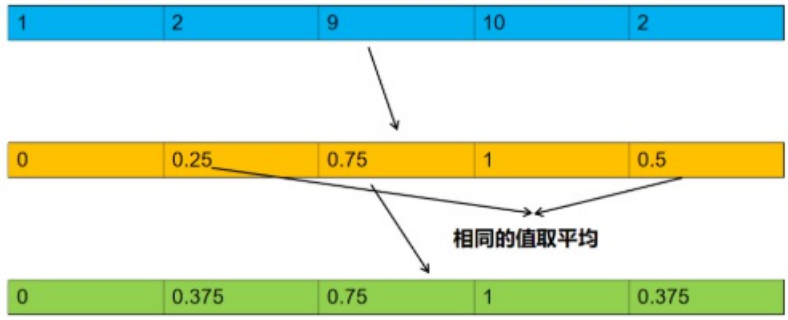
3.3 非线性变换

为什么要非线性转换

对于大多数数据挖掘算法来说，如果特征不服从或者近似服从标准正态分布（即，零均值、单位标准差的正态分布）的话，算法的表现会大打折扣。非线性转换就是将我们的特征映射到均匀分布或者高斯分布(即正态分布)。

映射到均匀分布

相比线性缩放，该方法不受异常值影响，它将数据映射到了零到一的均匀分布上，将最大的数映射为 `1`，最小的数映射为 `0`。其它的数按从小到大的顺序均匀分布在 `0` 到 `1` 之间，如有相同的数则取平均值，如数据为 `np.array([[1],[2],[3],[4],[5]])` 则经过转换为： `np.array([[0],[0.25],[0.5],[0.75],[1]])`，数据为 `np.array([[1],[2],[9],[10],[2]])` 则经过转换为： `np.array([[0],[0.375],[0.75],[1.0],[0.375]])`。第二个例子具体过程如下图：



在 `sklearn` 中使用 `QuantileTransformer` 方法实现，用法如下：

```
from sklearn.preprocessing import QuantileTransformer
import numpy as np

data = np.array([[1],[2],[3],[4],[5]])
quantile_transformer = QuantileTransformer(random_state=666)
data = quantile_transformer.fit_transform(data)

>>>data
array([[0.  ],
       [0.25],
       [0.5 ],
       [0.75],
       [1.  ]])
```

映射到高斯分布

映射到高斯分布是为了稳定方差，并最小化偏差。在最新版 `sklearn 0.20.x` 中 `PowerTransformer` 现在有两种映射方法， `Yeo-Johnson` 映射，公式如下：

$$x_i^{(\lambda)}=\begin{cases}[(x_i+1)^{\lambda}-1], if\lambda\neq 0, x_i\geq 0\\ \ln(x_i)+1, if\lambda=0, x_i\geq 0\\ -[(-x_i+1)^{2-\lambda}-1]/(2-\lambda), if\lambda\neq 2, x_i<0\\ -\ln(-x_i+1), if\lambda=2, x_i<0\end{cases}$$

`Box-Cox` 映射，公式如下：

$$x_i^{(\lambda)}=\begin{cases}\dfrac{x_i^{\lambda}-1}{\lambda}, if\lambda\neq 0\\ \ln(x_i), if\lambda=0\end{cases}$$

在 `sklearn 0.20.x` 中使用 `PowerTransformer` 方法实现，用法如下：

```
from sklearn.preprocessing import PowerTransformer
import numpy as np

data = np.array([[1],[2],[3],[4],[5]])
pt = PowerTransformer(method='box-cox', standardize=False)

data = pt.fit_transform(data)
```

学习平台使用的是 `sklearn 0.19.x`，通过对 `QuantileTransformer` 设置参数 `output_distribution='normal'` 实现映射高斯分布，用法如下：

```
from sklearn.preprocessing import QuantileTransformer
import numpy as np

data = np.array([[1],[2],[3],[4],[5]])
quantile_transformer = QuantileTransformer(output_distribution='normal',random_state=666)
data = quantile_transformer.fit_transform(data)
data = np.around(data,decimals=3)

>>>data
array([[-5.199],
       [-0.674],
       [ 0.   ],
       [ 0.674],
       [ 5.199]])
```

3.4 归一化

为什么使用归一化

归一化是缩放单个样本以具有单位范数的过程。归一化实质是一种线性变换，线性变换有很多良好的性质，这些性质决定了对数据改变后不会造成“失效”，反而能提高数据的表现，这些性质是归一化的前提。归一化能够加快模型训练速度，统一特征量纲，避免数值太大。值得注意的是，归一化是对每一个样本做转换，所以是对数据的每一行进行变换。而之前我们讲过的方法是对数据的每一列做变换。

L1范式归一化

`L1` 范式定义如下：

$$||x||_1 = \sum_{i=1}^{n} |x_i|$$

表示向量 `x` 中每个元素的绝对值之和。 `L1` 范式归一化就是将样本中每个特征除以特征的 `L1` 范式。

在 `sklearn` 中使用 `normalize` 方法实现，用法如下：

```
from sklearn.preprocessing import normalize

data = np.array([[-1,0,1],
                 [1,0,1],
                 [1,2,3]])
data = normalize(data,'l1')

>>>data
array([[-0.5  ,  0.   ,  0.5  ],
       [ 0.5  ,  0.   ,  0.5  ],
       [ 0.167,  0.333,  0.5  ]])
```

L2范式归一化

`L2` 范式定义如下：

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

表示向量元素的平方和再开平方根。 `L2` 范式归一化就是将样本中每个特征除以特征的 `L2` 范式。

在 `sklearn` 中使用 `normalize` 方法实现，用法如下：

```
from sklearn.preprocessing import normalize

data = np.array([[-1,0,1],
                 [1,0,1],
```

```
                [1,2,3]])
data = normalize(data,'l2')

>>>data
array([[-0.707,  0.   ,  0.707],
       [ 0.707,  0.   ,  0.707],
       [ 0.267,  0.535,  0.802]])
```

3.5 离散值编码

LabelEncoder

在数据挖掘中，特征经常不是数值型的而是分类型的。举个例子，一个人可能有 `["male", "female"]`， `["from Europe", "from US", "from Asia"]`， `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]` 等分类的特征。这些特征能够被有效地编码成整数，比如 `["male", "from US", "uses Internet Explorer"]` 可以被表示为 `[0, 1, 3]`， `["female", "from Asia", "uses Chrome"]` 表示为 `[1, 2, 1]`。

在 `sklearn` 中，通过 `LabelEncoder` 来实现：

```
from sklearn.preprocessing import LabelEncoder

label = ['male','female']
int_label = LabelEncoder()
label = int_label.fit_transform(label)

>>>label
array([1, 0])
```

OneHotEncoder

这种整数特征表示并不能在 `sklearn` 的估计器中直接使用，因为这样的连续输入，估计器会认为类别之间是有序的，但实际却是无序的。如将 `male,female`，转换为 `1,0`。 `1` 比 `0` 要大，机器就会把这个关系考虑进去，而 `male,female` 之间是没有这样的关系的。所以我们需要使用另外一种编码方式， `OneHot` 编码。

`OneHot` 编码其实非常简单，就是在将原来的特征展开成一个二进制列，假设 `sex` 这个特征有两种取值，分别为: `male` 和 `female`。数据如下：

id	sex
1	male
2	male
3	female
4	male
5	female

那么经过 `OneHot` 编码之后，数据会变成如下形式(原来数据中 `sex` 为 `male` 的在编码后 `sex_male` 的值为 `1`， `sex` 为 `female` 的在编码后 `sex_female` 的值为 `1`)：

id	sex_male	sex_female
1	1	0
2	1	0
3	0	1
4	1	0

5	0	1

你会发现，经过 `OneHot` 编码后， `sex` 特征变成了无序的二进制特征。当然，在 `sklearn` 中已经为我们提供了 `OneHot` 编码的接口。可以通过 `OneHotEncoder` 来实现，使用方法如下：

```
import numpy as np
from sklearn.preprocessing import OneHotEncoder

label = np.array([1,0])
label = np.array(label).reshape(len(label),1)#先将X组织成（sample，feature）的格式

# 实例化OneHotEncoder对象
onehot_label = OneHotEncoder()
# 对数据进行编码
label = onehot_label.fit_transform(label).toarray()

>>>label
array([[0., 1.],
       [1., 0.]])
```

3.6 生成多项式特征

为什么需要多项式特征

在数据挖掘中，获取数据的代价经常是非常高昂的。所以有时就需要人为的制造一些特征，并且有的特征之间是有关联的。生成多项式特征可以轻松的为我们获取更多的数据，并获得特征的更高维度和互相间关系的项且引入了特征之间的非线性关系，可以有效的增加模型的复杂度。

PolynomialFeatures

在 `sklearn` 中通过 `PolynomialFeatures` 方法来生成多项式特征，使用方法如下：

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

data = np.arange(6).reshape(3, 2)
poly = PolynomialFeatures(2)#生成二项式特征
data = poly.fit_transform(data)

>>>data
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

特征转换情况如下：

$$(x_1, x_2) \longrightarrow (1, x_1, x_2, x_1^2, x_1x_2, x_2^2)$$

在一些情况下，只需要特征间的交互项，这可以通过设置 `interaction_only=True` 来得到:

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

data = np.arange(6).reshape(3, 2)
poly = PolynomialFeatures(degree=2, interaction_only=True)#degree=n表示生成n项式特征，只需要特征之间交互
data = poly.fit_transform(data)

>>>data
array([[ 1.,  0.,  1.,  0.],
       [ 1.,  2.,  3.,  6.],
       [ 1.,  4.,  5., 20.]])
```

特征转换情况如下：

$$(x_1, x_2) \longrightarrow (1, x_1, x_2, x_1x_2)$$

3.7 估算缺失值

为什么要估算缺失值

由于各种原因，真实世界中的许多数据集都包含缺失数据，这类数据经常被编码成空格、 `NaNs` ，或者是其他的占位符。但是这样的数据集并不能被 `sklearn` 学习算法兼容，因为大多的学习算法都默认假设数组中的元素都是数值，因而所有的元素都有自己的意义。 使用不完整的数据集的一个基本策略就是舍弃掉整行或整列包含缺失值的数据。但是这样就付出了舍弃可能有价值数据（即使是不完整的 ）的代价。 处理缺失数值的一个更好的策略就是从已有的数据推断出缺失的数值。

Imputer

`sklearn` 中使用 `Imputer` 方法估算缺失值，使用方法如下：

```
from sklearn.preprocessing import Imputer

data = [[np.nan, 2], [6, np.nan], [7, 4],[np.nan,4]]
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)#缺失值为nan，沿着每一列，使用平均值来代替缺失值
data = imp.fit_transform(data)

>>>data
array([[6.5       , 2.        ],
       [6.        , 3.33333333],
       [7.        , 4.        ],
       [6.5       , 4.        ]])
```

其中 `strategy` 参数用来选择代替缺失值方法：

```
`mean`表示使用平均值代替缺失值
`median`表示使用中位数代替缺失值
`most_frequent`表示使用出现频率最多的值代替缺失值
```

`missing_values` 参数表示何为缺失值：

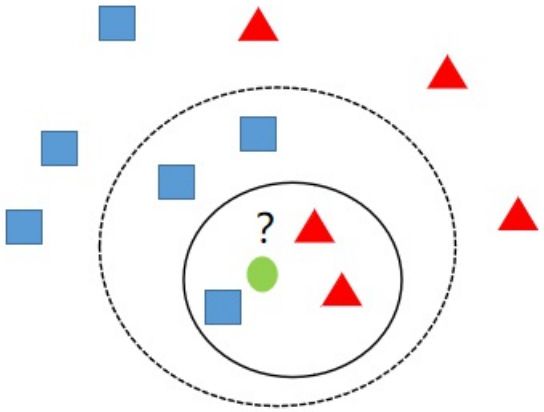
```
`NaN`表示`np.nan`为缺失值
`0`表示`0`为缺失值
```

第四章 **k-**近邻

4.1 k-近邻算法思想

k-近邻 （k-nearest neighbor ,knn） 是一种分类与回归的方法。我们这里只讨论用来分类的 knn 。所谓 k 最近邻，就是 k 个最近的邻居的意思，说的是每个样本都可以用它最近的 k 个邻居来代表。

knn 算法的核心思想是如果一个样本在特征空间中的 k 个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。 knn 方法在类别决策时，只与极少量的相邻样本有关。



如上图，当 k=3 时离绿色的圆最近的三个样本中，有两个红色的三角形，一个蓝色的正方形，则此时绿色的圆应该分为红色的三角形这一类。而当 k=5 时，离绿色的圆最近的五个样本中，有两个红色的三角形，三个蓝色的正方形，则此时绿色的圆应该分为蓝色的正方形这一类。

用一句话来总结 knn 算法的思想就是近朱者赤近墨者黑。而且 knn 是一种简单而有效的机器学习算法。

4.2 k-近邻算法原理

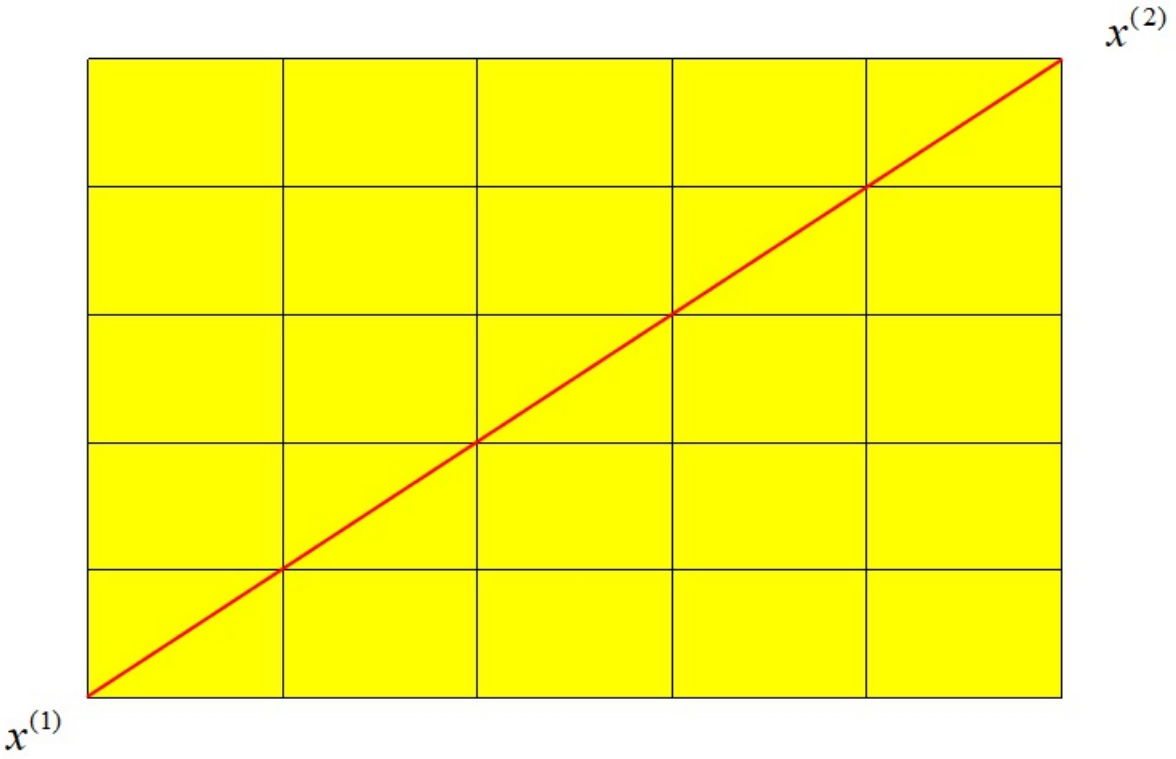
我们已经知道，如何判别一个样本属于哪个类型，主要是看离它最近的几个样本中哪个类型的数量最多，则该样本属于数量最多的类型。这里，存在两个问题：

- 何为最近
- 如果有两个类型的样本数一样且最多，那么最终该样本应该属于哪个类型

距离度量

关于何为最近，大家应该自然而然就会想到可以用两个样本之间的距离大小来衡量，我们常用的有两种距离：

- 欧氏距离：欧氏距离是最容易直观理解的距离度量方法，我们小学、初中和高中接触到的两个点在空间中的距离一般都是指欧氏距离。



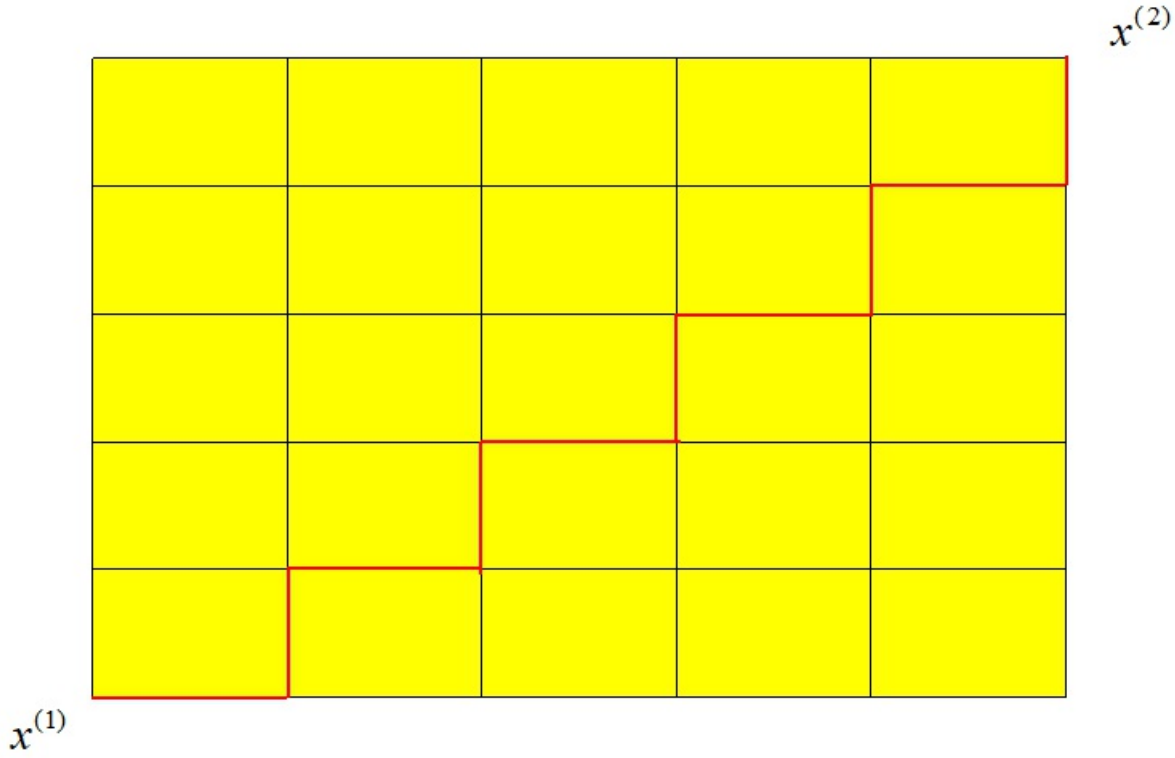
二维平面上欧式距离计算公式：

$$d_{12}=\sqrt{(x_1^{(1)}-x_1^{(2)})^2+(x_2^{(1)}-x_2^{(2)})^2}$$

`n` 维平面上欧氏距离计算公式：

$$d_{12}=\sqrt{\sum_{i=1}^{n}(x_i^{(1)}-x_i^{(2)})^2}$$

- 曼哈顿距离：顾名思义，在曼哈顿街区要从一个十字路口开车到另一个十字路口，驾驶距离显然不是两点间的直线距离。这个实际驾驶距离就是“曼哈顿距离”。曼哈顿距离也称为“城市街区距离”。



二维平面上曼哈顿距离计算公式：

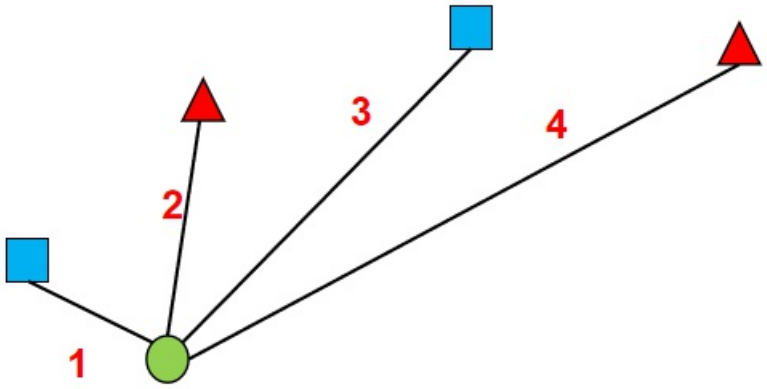
$$d_{12} = |x_1^{(1)} - x_1^{(2)}| + |x_2^{(1)} - x_2^{(2)}|$$

`n` 维平面上曼哈顿计算公式：

$$d_{12} = \sum_{i=1}^{n} |x_i^{(1)} - x_i^{(2)}|$$

加权投票

`knn` 算法最后决定样本属于哪个类别，其实好比就是在投票，哪个类别票数多，则该样本属于哪个类别。而如果出现票数相同的情况，我们可以给每一票加上一个权重，用来表示每一票的重要性，这样就可以解决票数相同的问题了。很明显，距离越近的样本所投的一票应该越重要，此时我们可以将距离的倒数作为权重赋予每一票。



$$\frac{1}{1}+\frac{1}{3}>\frac{1}{2}+\frac{1}{4}$$

如上图，虽然蓝色正方形与红色三角形数量一样，但是根据加权投票的规则，绿色的圆应该属于蓝色正方形这个类别。

4.3 k-近邻算法流程

`knn` 算法不需要训练模型，只是根据离样本最近的几个样本类型来判别该样本类型，所以流程非常简单：

- 1.计算出新样本与每一个样本的距离
- 2.找出距离最近的 `k` 个样本
- 3.根据加权投票规则得到新样本的类别

4.4 动手实现k-近邻

`knn` 算法实现 `python` 代码如下：

```
#encoding=utf8
import numpy as np

def knn_clf(k,train_feature,train_label,test_feature):
    '''
    input:
        k(int):最近邻样本个数
        train_feature(ndarray):训练样本特征
        train_label(ndarray):训练样本标签
        test_feature(ndarray):测试样本特征
    output:
        predict(ndarray):测试样本预测标签
    '''
    #初始化预测结果
    predict = np.zeros(test_feature.shape[0],).astype('int')
    #对测试集每一个样本进行遍历
    for i in range(test_feature.shape[0]):
        #测试集第i个样本到训练集每一个样本的距离
        distance = np.sqrt(np.power(np.tile(test_feature[i],(train_feature.shape[0],1))-train_feature,2).sum(
axis=1))
        #最近的k个样本的距离
        distance_k = np.sort(distance)[:k]
        #最近的k个样本的索引
        nearest = np.argsort(distance)[:k]
        #最近的k个样本的标签
        topK = [train_label[i] for i in nearest]
        #初始化进行投票的字典，字典的键为标签，值为投票分数
        votes = {}
        #初始化最大票数
        max_count = 0
        #进行投票
        for j,label in enumerate(topK):
            #如果标签在字典的键中则投票计分
            if label in votes.keys():
                votes[label] += 1/(distance_k[j]+1e-10)#防止分母为0
                #如果评分最高则将预测值更新为对应标签
                if votes[label] > max_count:
                    max_count = votes[label]
                    predict[i] = label
            #如果标签不在字典中则将标签加入字典的键，同时计入相应的分数
            else:
                votes[label] = 1/(distance_k[j]+1e-10)
                if votes[label] > max_count:
                    max_count = votes[label]
                    predict[i] = label
    return predict
```

4.5 实战案例

手写数字数据

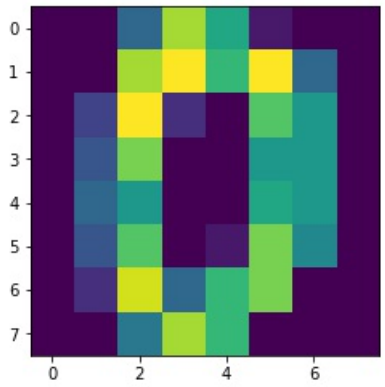
手写数字数据集一共有 `1797` 个样本，每个样本有 `64` 个特征。每个特征的值为 `0-255` 之间的像素，我们的任务就是根据这 `64` 个特征值识别出该数字属于 `0-9` 十个类别中的哪一个。

我们可以使用 `sklearn` 直接对数据进行加载，代码如下：

```
from sklearn.datasets import load_digits
#加载手写数字数据集
digits = load_digits()
#获取数据特征与标签
x,y = digits .data,digits .target
```

当然，每一个样本就是一个数字，我们可以把它还原为 `8x8` 的大小进行查看：

```
import matplotlib.pyplot as plt

img = x[0].reshape(8,8)
plt.imshow(img)
```



然后我们划分出训练集与测试集，训练集用来训练模型，测试集用来检测模型性能。代码如下：

```
from sklearn.model_selection import train_test_split
#划分训练集测试集，其中测试集样本数为整个数据集的20%
train_feature,test_feature,train_label,test_label = train_test_split(x,y,test_size=0.2,random_state=666)
```

进行识别

接下来就只需要调用之前实现的 `knn_clf` 方法就可以对测试集的手写数字进行识别了：

```
predict = knn_clf(3,train_feature,train_label,test_feature)
predict
>>>array([1, 5, 0, 7, 1, 0, 6, 1, 5, 4, 9, 2, 7, 8, 4, 6, 9, 3, 7, 4, 7, 1,
       8, 6, 0, 9, 6, 1, 3, 7, 5, 9, 8, 3, 2, 8, 8, 1, 1, 0, 7, 9, 0, 0,
       8, 7, 2, 7, 4, 3, 4, 3, 4, 0, 4, 7, 0, 5, 5, 5, 2, 1, 7, 0, 5, 1,
```

```
       8, 3, 3, 4, 0, 3, 7, 4, 3, 4, 2, 9, 7, 3, 2, 5, 3, 4, 1, 5, 5, 2,
       9, 2, 2, 2, 2, 7, 0, 8, 1, 7, 4, 2, 3, 8, 2, 3, 3, 0, 2, 9, 9, 2,
       3, 2, 8, 1, 1, 9, 1, 2, 0, 4, 8, 5, 4, 4, 7, 6, 7, 6, 6, 1, 7, 5,
       6, 3, 8, 3, 7, 1, 8, 5, 3, 4, 7, 8, 5, 0, 6, 0, 6, 3, 7, 6, 5, 6,
       2, 2, 2, 3, 0, 7, 6, 5, 6, 4, 1, 0, 6, 0, 6, 4, 0, 9, 3, 8, 1, 2,
       3, 1, 9, 0, 7, 6, 2, 9, 3, 5, 3, 4, 6, 3, 3, 7, 4, 9, 2, 7, 6, 1,
       6, 8, 4, 0, 3, 1, 0, 9, 9, 9, 0, 1, 8, 6, 8, 0, 9, 5, 9, 8, 2, 3,
       5, 3, 0, 8, 7, 4, 0, 3, 3, 3, 6, 3, 3, 2, 9, 1, 6, 9, 0, 4, 2, 2,
       7, 9, 1, 6, 7, 6, 3, 9, 1, 9, 3, 4, 0, 6, 4, 8, 5, 3, 6, 3, 1, 4,
       0, 4, 4, 8, 7, 9, 1, 5, 2, 7, 0, 9, 0, 4, 4, 0, 1, 0, 6, 4, 2, 8,
       5, 0, 2, 6, 0, 1, 8, 2, 0, 9, 5, 6, 2, 0, 5, 0, 9, 1, 4, 7, 1, 7,
       0, 6, 6, 8, 0, 2, 2, 6, 9, 9, 7, 5, 1, 7, 6, 4, 6, 1, 9, 4, 7, 1,
       3, 7, 8, 1, 6, 9, 8, 3, 2, 4, 8, 7, 5, 5, 6, 9, 9, 8, 5, 0, 0, 4,
       9, 3, 0, 4, 9, 4, 2, 5])
```

再根据测试集标签即真实分类结果，计算出正确率：

```
acc = np.mean(predict==test_label)
acc
>>>0.994
```

可以看到，使用 `knn` 对手写数字进行识别，正确率能达到 `99%` 以上。

第五章 线性回归

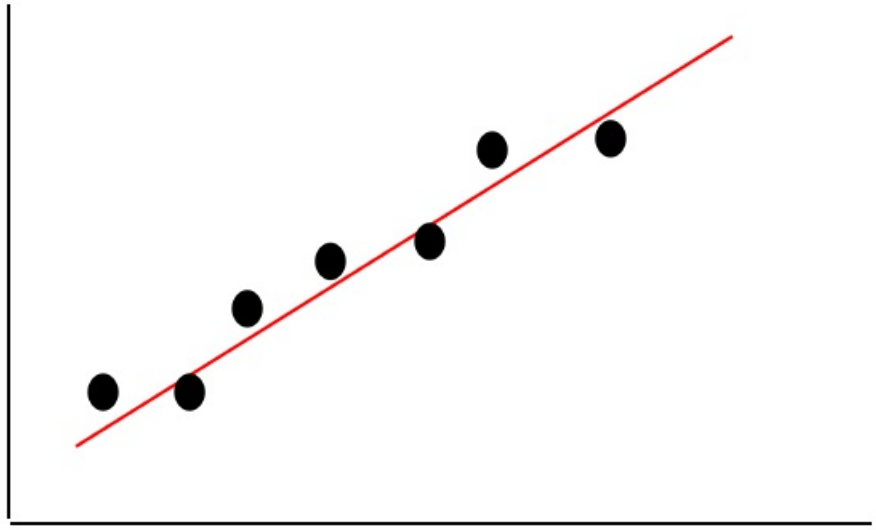
5.1 线性回归算法思想

简单线性回归

在生活中，我们常常能碰到这么一种情况，一个变量会跟着另一个变量的变化而变化，如圆的周长与半径的关系，当圆的半径确定了，那么周长也就确定了。还有一种情况就是，两个变量之间看似存在某种关系，但又没那么确定，如青少年的身高与体重，他们存在一种近似的线性关系：

`身高/cm = 体重/kg +105`

但是，并不是每个青少年都符合这个公式，只能说每个青少年的身高体重都存在这么一种近似的线性关系。这就是其实就是简单的线性回归，那么，到底什么是线性回归呢？假如我们将青少年的身高和体重值作为坐标，不同人的身高体重就会在平面上构成不同的坐标点，然后用一条直线，尽可能的去拟合这些点，这就是简单的线性回归。



简单的线性回归模型如下：

$$y = wx + b$$

其中 `x` 表示特征值(如：体重值)， `w` 表示权重， `b` 表示偏置， `y` 表示标签(如：身高值)。

多元线性回归

简单线性回归中，一个变量跟另一个变量的变化而变化，但是生活中，还有很多变量，可能由多个变量的变化决定着它的变化，比如房价，影响它的因素可能有：房屋面积、地理位置等等。如果我们要给它们建立出近似的线性关系，这就是多元线性回归，多元线性回归模型如下：

$$y = b + w_1x_1 + w_2x_2 + ... + w_nx_n$$

其中，x_i表示第 `i` 个特征，w_i表示第 `i` 个特征对于的权重， `b` 表示偏置， `y` 表示标签。

5.2 线性回归算法原理

线性回归训练流程

我们已经知道线性回归模型如下：

$$y = b + w_1x_1 + w_2x_2 + ... + w_nx_n$$

为了方便，我们稍微将模型进行变换：

$$y = w_0x_0 + w_1x_1 + w_2x_2 + ... + w_nx_n$$

其中 `x0=1` , `w0=b` ,通过向量化公式可写成如下形式：

$$Y = X.W$$

$$W = (w_0, w_1, ..., w_n)$$

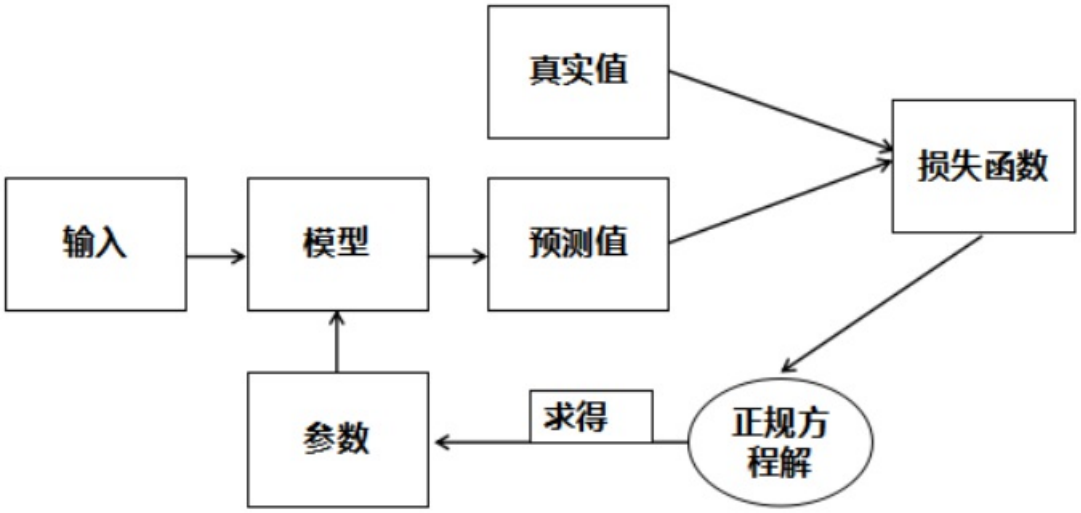
$$X = (1, x_1, ..., x_n)$$

而我们的目的就是找出能够正确预测的多元线性回归模型，即找出正确的 `W` （即权重与偏置）。那么如何寻找呢？通常在监督学习里面都会使用这么一个套路，构造一个损失函数，用来衡量真实值与预测值之间的差异，然后将问题转化为最优化损失函数。既然损失函数是用来衡量真实值与预测值之间的差异那么很多人自然而然的想到了用所有真实值与预测值的差的绝对值来表示损失函数。不过带绝对值的函数不容易求导，所以采用 `MSE` (均方误差)作为损失函数，公式如下：

$$loss = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - p^{(i)})^2$$

其中 `p` 表示预测值， `y` 表示真实值， `m` 为样本总个数， `i` 表示第 `i` 个样本。最后，我们再使用正规方程解来求得我们所需要的参数。

线性回归模型训练流程图如下：



正规方程解

对线性回归模型，假设训练集中 `m` 个训练样本，每个训练样本中有 `n` 个特征，可以使用矩阵的表示方法，预测函数可以写为：

$$Y = X.W$$

其损失函数可以表示为

$$loss = \frac{1}{m}(Y - X.W)^T(Y - X.W)$$

其中，标签 `Y` 为 `m` 行 `1` 列的矩阵，训练特征 `X` 为 `m` 行 `(n+1)` 列的矩阵，回归系数 `W` 为 `(n+1)` 行 `1` 列的矩阵，对 `W` 求导，并令其导数为零可解得：

$$W = (X^T X)^{-1} X^T Y$$

这个就是正规方程解，我们可以通过正规方程解直接求得我们所需要的参数。

5.3 线性回归算法流程

我们最终的目的是通过训练出来的线性回归模型对测试集数据进行预测，算法实现流程如下：

- 1.将 `x0=1` 加入训练数据
- 2.使用正规方程解求得参数
- 3.将 `x0=1` 加入测试数据
- 4.对测试集数据进行预测

5.4 动手实现线性回归

线性回归 `python` 实现代码如下：

```python
#encoding=utf8
import numpy as np

def lr(train_feature,train_label,test_feature):
    '''
    input:
        train_feature(ndarray):训练样本特征
        train_label(ndarray):训练样本标签
        test_feature(ndarray):测试样本特征
    output:
        predict(ndarray):测试样本预测标签
    '''
    #将x0=1加入训练数据
    train_x = np.hstack([np.ones((len(train_feature),1)),train_feature])
    #使用正规方程解求得参数
    theta =np.linalg.inv(train_x.T.dot(train_x)).dot(train_x.T).dot(train_label)
    #将x0=1加入测试数据
    test_x = np.hstack([np.ones((len(test_feature),1)),test_feature])
    #求得测试集预测标签
    predict = test_x.dot(theta)
    return predict
```

5.5 实战案例

波士顿房价数据

波士顿房价数据集共有 `506` 条房价数据，每条数据包括对指定房屋的 `13` 项数值型特征和目标房价组成。我们需要通过数据特征来对目标房价进行预测。

数据集中部分数据与标签如下图所示：

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21

	0
0	24.0
1	21.6
2	34.7
3	33.4
4	36.2
5	28.7

`sklearn` 中已经提供了波士顿房价数据集的相关接口，想要使用该数据集可以使用如下代码：

```
from sklearn import datasets
#加载波士顿房价数据集
boston = datasets.load_boston()
#X表示特征，y表示目标房价
x = boston.data
y = boston.target
```

然后再对数据集进行划分：

```
from sklearn.model_selection import train_test_split
#划分训练集测试集，所有样本的20%作为测试集
train_feature,test_feature,train_label,test_label = train_test_split(x,y,test_size=0.2,random_state=666)
```

进行预测

同样的只需要调用之前实现线性回归方法就可以对测试集的波士顿房价数据进行预测了：

```
predict = lr(train_feature,train_label,test_feature)
>>>predict
array([27.14328365, 23.03653632, 27.00098113, 34.67246356, 22.9249281 ,
       21.27666411, 15.67682012, 23.71041177, 24.9170328 , 18.94485146,
```

```
        4.21475157, 24.91145159, 20.98995302, 18.43508891, 24.17666486,
       26.84239278, 27.83397467, 13.52699359, 18.45498398, 28.42388411,
       30.59256907, 13.41724252,  8.12085396, 35.51572129, 25.67615918,
       17.16601994, 20.37433719, 13.09756854, 34.29369038, 23.73452722,
       39.80575322,  8.23996654, 24.79976309, 17.93534789, 23.166615  ,
       19.77561659, 35.15958711, 35.62614752, 21.48402467, 13.53651885,
       23.8764859 , 22.76090085, 27.69433621, 18.25312903, 28.24166439,
       11.37889658, 27.10532052, 32.76787747, 29.42762069, 24.90135914,
       27.29432351, 33.19296658, 26.14048342, 23.62626694, 27.59078519,
       20.00241919, 14.46427082, 20.0119397 , 19.81015781, 13.93309224,
       20.96227953, 25.93383085, 30.17587814, 18.06438076, 12.03215906,
       11.3801673 , 26.81093528, 22.56148123, 22.95599483, 25.79865129,
       10.10532755, 33.63114297, 17.81932257, 17.21896388, 39.33351986,
       14.91994896, 18.19524145, 24.94373123, 20.09101825, 31.48389087,
       32.8430831 , 23.95919903,  9.77345135, 31.55307878, 30.55370904,
       23.20332797, 21.90050123, 13.5557125 , 18.27957707, 25.0240593 ,
       19.54159097, 36.39430746, 24.02473259, 33.08973723, 21.71311184,
       17.37919862, 26.67885309, 27.42896672, 13.1943355 ,  0.57642556,
       19.69396665, 14.18869608])
```

衡量回归的性能指标

对于分类问题，我们可以使用正确率来衡量模型的性能好坏，很明显，回归问题并不能使用正确率来衡量，那么，回归问题可以使用哪些指标用来评估呢？

MSE

`MSE （Mean Squared Error）` 叫做均方误差,公式如下：

$$mse = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - p^{(i)})^2$$

其中 `yi` 表示第 `i` 个样本的真实标签， `pi` 表示第 `i` 个样本的预测标签。线性回归的目的就是让损失函数最小。那么，模型训练出来了，我们再测试集上用损失函数来评估也是可以的。

RMSE

`RMSE（Root Mean Squard Error）` 均方根误差，公式如下：

$$rmse = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - p^{(i)})^2}$$

`RMSE` 其实就是 `MSE` 开个根号。有什么意义呢？其实实质是一样的。只不过用于数据更好的描述。

例如：要做房价预测，每平方是万元，我们预测结果也是万元。那么差值的平方单位应该是千万级别的。那我们不太好描述自己做的模型效果。怎么说呢？我们的模型误差是多少千万？于是干脆就开个根号就好了。我们误差的结果就跟我们数据是一个级别的了，在描述模型的时候就说，我们模型的误差是多少万元。

MAE

`MAE` (平均绝对误差)，公式如下：

$$mae = \frac{1}{m}\sum_{i=1}^{m}|y^{(i)} - p^{(i)}|$$

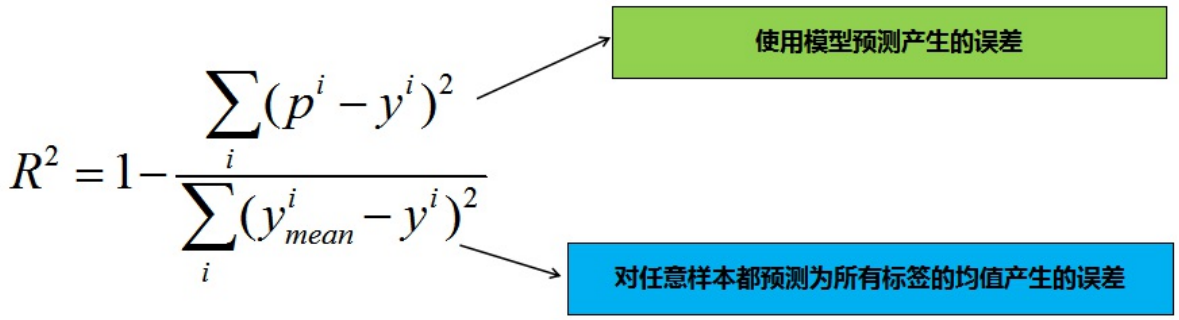
`MAE` 虽然不作为损失函数，确是一个非常直观的评估指标，它表示每个样本的预测标签值与真实标签值的 `L1` 距离。

R-Squared

上面的几种衡量标准针对不同的模型会有不同的值。比如说预测房价 那么误差单位就是万元。数字可能是 `3`， `4`， `5` 之类的。那么预测身高就可能是 `0.1`， `0.6` 之类的。没有什么可读性，到底多少才算好呢？不知道，那要根据模型的应用场景来。 看看分类算法的衡量标准就是正确率，而正确率又在 `0~1` 之间，最高百分之百。最低 `0` 。那么回归有没有这样的衡量标准呢？ `R-Squared` 就是这么一个指标，公式如下：

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(p^{(i)} - y^{(i)})^2}{\sum_{i=1}^{m}(y_{mean}^{(i)} - y^{(i)})^2}$$

为什么这个指标会有刚刚我们提到的性能呢？我们分析下公式：



其实分子表示的是模型预测时产生的误差，分母表示的是对任意样本都预测为所有标签均值时产生的误差，由此可知：

- 1.当我们的模型不犯任何错误时，取最大值 `1` 。
- 2.当我们的模型性能跟基模型性能相同时，取 `0` 。
- 3.如果为负数，则说明我们训练出来的模型还不如基准模型，此时，很有可能我们的数据不存在任何线性关系。

其中，基准模型以标签的均值作为所有样本的预测结果，具有极大的误差。

这里使用 `python` 实现了 `MSE` ， `R-Squared` 方法，代码如下：

```
import numpy as np

#mse
def mse_score(y_predict,y_test):
    mse = np.mean((y_predict-y_test)**2)
    return mse
#r2
def r2_score(y_predict,y_test):
    '''
    input:y_predict(ndarray):预测值
          y_test(ndarray):真实值
    output:r2(float):r2值
```

```
    '''
    r2 = 1 - mse_score(y_predict,y_test)/np.var(y_test)
    return r2
```

我们可以根据求得的预测值，计算出 `MSE` 值与 `R-Squared` 值:

```
mse = mse_score(predict,test_label)
mse
>>>27.22
r2 = r2_score(predict,test_label)
r2
>>>0.63
```

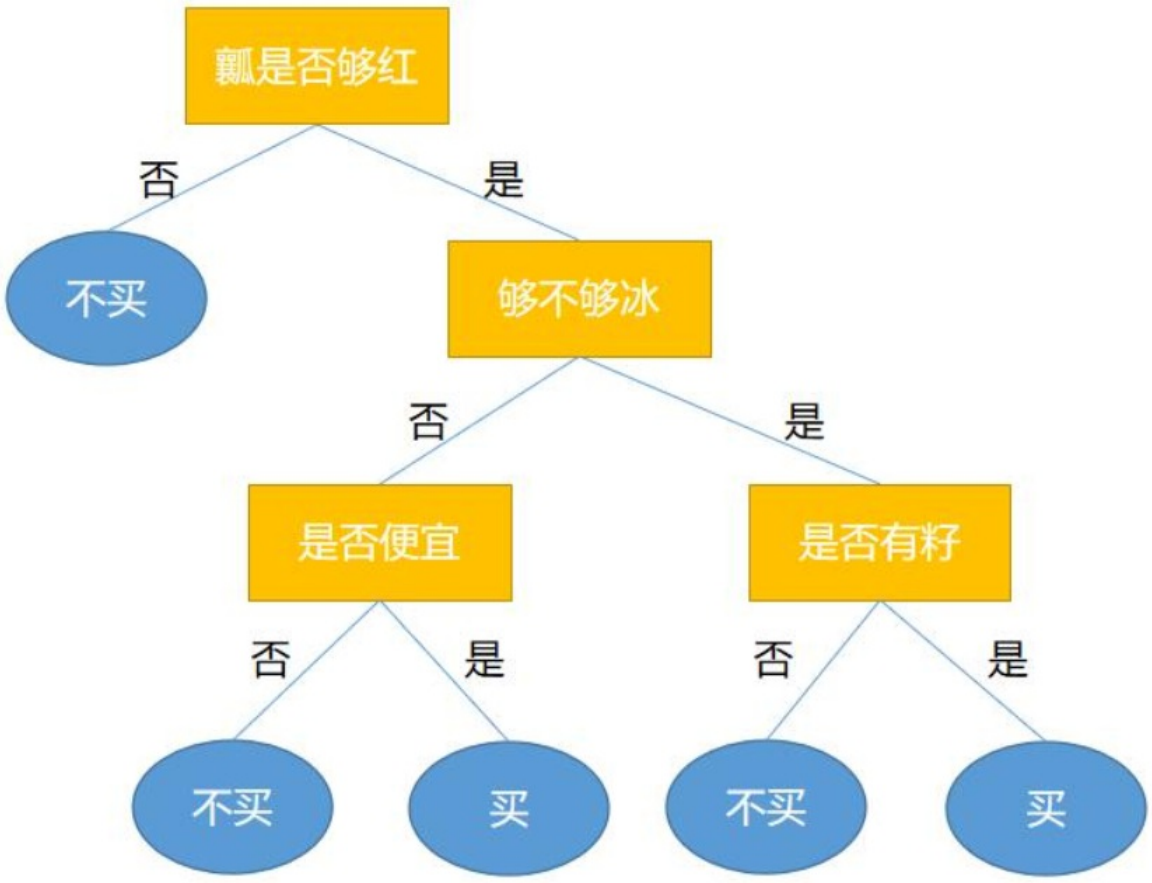
第六章 决策树

6.1 决策树算法思想

决策树是一种可以用于分类与回归的机器学习算法，但主要用于分类。用于分类的决策树是一种描述对实例进行分类的树形结构。决策树由结点和边组成，其中结点分为内部结点和叶子结点，内部结点表示一个特征或者属性，叶子结点表示标签。

决策树说通俗点就是一棵能够替我们做决策的树，或者说是我们人类在要做决策时脑回路的一种表现形式，我们可以从下面这个例子来了解决策树是什么。

在炎热的夏天，没有什么比冰镇后的西瓜更能令人感到心旷神怡的了。现在我要去水果店买西瓜，但怎样我才会买这个西瓜呢？那么，有可能我会有以下这个决策过程：



假设现在水果店里有 3 个西瓜，它们的属性如下：

编号	瓤是否够红	够不够冰	是否便宜	是否有籽
1	是	否	是	否
2	是	是	否	是
3	否	是	是	否

那么根据我的决策过程我会买 1 和 2 号西瓜。这个帮助我选择西瓜的过程，就是一个决策树。由之前介绍的知识可以知道，黄色部分为内部节点，蓝色部分为叶子节点。

6.2 决策树算法原理

我们已经知道，构造一棵决策树其实就是根据数据的特征(内部节点)对数据一步一步的进行划分，从而达到分类的目的。但是，每一步根据哪个特征来进行划分呢？这个就是构造决策树的关键。其实构造决策树时会遵循一个指标，有的是按照信息增益来构建，如**ID3**算法；有的是信息增益率来构建，如**C4.5**算法；有的是按照基尼系数来构建的，如**CART**算法。但不管是使用哪种构建算法，决策树的构建过程通常都是一个递归选择最优特征，并根据特征对训练集进行分割，使得对各个子数据集有一个最好的分类的过程。这里我们以**ID3**算法为例，详细介绍构建决策树相关知识。

信息熵

信息是个很抽象的概念。人们常常说信息很多，或者信息较少，但却很难说清楚信息到底有多少。比如一本五十万字的中文书到底有多少信息量。

直到 `1948` 年，香农提出了**“信息熵”**的概念，才解决了对信息的量化度量问题。信息熵这个词是香农从热力学中借用过来的。热力学中的热熵是表示分子状态混乱程度的物理量。香农用信息熵的概念来描述信源的不确定度。信源的不确定性越大，信息熵也越大。

从机器学习的角度来看，信息熵表示的是信息量的期望值。如果数据集中的数据需要被分成多个类别，则信息量 $I(x_i)$的定义如下：

其中x_i表示多个类别中的第 `i` 个类别，$p(x_i)$表示概率：

$$I(X_i) = -log_2 p(x_i)$$

由于信息熵是信息量的期望值，所以信息熵$H(X)$的定义如下(其中 `n` 为数据集中类别的数量)：

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i)$$

从这个公式也可以看出，如果概率是 `0` 或者是 `1` 的时候，熵就是 `0` 。（因为这种情况下随机变量的不确定性是最低的），那如果概率是 `0.5` 也就是五五开的时候，此时熵达到最大，也就是 `1` 。（就像扔硬币，你永远都猜不透你下次扔到的是正面还是反面，所以它的不确定性非常高）。所以呢，熵越大，不确定性就越高。

条件熵

在实际的场景中，我们可能需要研究数据集中某个特征等于某个值时的信息熵等于多少，这个时候就需要用到条件熵。条件熵 `H(Y|X)` 表示特征 `X` 为某个值的条件下，类别为 `Y` 的熵。条件熵的计算公式如下：

$$H(Y|X) = \sum_{i=1}^{n} p_i H(Y|X = x_i)$$

信息增益

现在已经知道了什么是熵，什么是条件熵。接下来就可以看看什么是信息增益了。所谓的信息增益就是表示我已知条件 `X` 后能得到信息 `Y` 的不确定性的减少程度。

就好比，我在玩读心术。你心里想一件东西，我来猜。我已开始什么都没问你，我要猜的话，肯定是瞎猜。这个时候我的熵就非常高。然后我接下来我会去试着问你是非题，当我问了是非题之后，我就能减小猜测你心中想到的东西的范围，这样其实就是减小了我的熵。那么我熵的减小程度就是我的信息增益。

所以信息增益如果套上机器学习的话就是，如果把特征 `A` 对训练集 `D` 的信息增益记为 `g(D, A)` 的话，那么 `g(D, A)` 的计算公式就是：

$$g(D, A) = H(D) - H(D, A)$$

为了更好的解释熵，条件熵，信息增益的计算过程，下面通过示例来描述。假设我现在有这一个数据集，第一列是编号，第二列是性别，第三列是活跃度，第四列是客户是否流失的标签（ `0:` 表示未流失， `1:` 表示流失）。

编号	性别	活跃度	是否流失
1	男	高	0
2	女	中	0
3	男	低	1
4	女	高	0
5	男	高	0
6	男	中	0
7	男	中	1
8	女	中	0
9	女	低	1
10	女	中	0
11	女	高	0
12	男	低	1
13	女	低	1
14	男	高	0
15	男	高	0

假如要算性别和活跃度这两个特征的信息增益的话，首先要先算总的熵和条件熵。总的熵其实非常好算，就是把标签作为随机变量 `X` 。上表中标签只有两种（ `0` 和 `1` ）因此随机变量 `X` 的取值只有 `0` 或者 `1` 。所以要计算熵就需要先分别计算标签为 `0` 的概率和标签为 `1` 的概率。从表中能看出标签为 `0` 的数据有 `10` 条，所以标签为 `0` 的概率等于 `2/3` 。标签为 `1` 的概率为 `1/3` 。所以熵为：

$$-\frac{1}{3} * log(\frac{1}{3}) - \frac{2}{3} * log(\frac{2}{3}) = 0.9182$$

接下来就是条件熵的计算，以性别为男的熵为例。表格中性别为男的数据有 `8` 条，这 `8` 条数据中有 `3` 条数据的标签为 `1` ，有 `5` 条数据的标签为 `0` 。所以根据条件熵的计算公式能够得出该条件熵为：

$$-\frac{3}{8} * log(\frac{3}{8}) - \frac{5}{8} * log(\frac{5}{8}) = 0.9543$$

根据上述的计算方法可知，总熵为：

$$-\frac{5}{15}*log(\frac{5}{15})-\frac{10}{15}*log(\frac{10}{15})=0.9182$$

性别为男的熵为：

$$-\frac{3}{8}*log(\frac{3}{8})-\frac{5}{8}*log(\frac{5}{8})=0.9543$$

性别为女的熵为：

$$-\frac{2}{7}*log(\frac{2}{7})-\frac{5}{7}*log(\frac{5}{7})=0.8631$$

活跃度为低的熵为：

$$-\frac{4}{4}*log(\frac{4}{4})=0$$

活跃度为中的熵为：

$$-\frac{1}{5}*log(\frac{1}{5})-\frac{4}{5}*log(\frac{4}{5})=0.7219$$

活跃度为高的熵为：

$$-0-\frac{6}{6}*log(\frac{6}{6})=0$$

现在有了总的熵和条件熵之后就能算出性别和活跃度这两个特征的信息增益了。

性别的信息增益=总的熵-(8/15)性别为男的熵-(7/15)性别为女的熵=0.0064

活跃度的信息增益=总的熵-(6/15)活跃度为高的熵-(5/15)活跃度为中的熵-(4/15)活跃度为低的熵=0.6776

那信息增益算出来之后有什么意义呢？回到读心术的问题，为了我能更加准确的猜出你心中所想，我肯定是问的问题越好就能猜得越准！换句话来说我肯定是要想出一个信息增益最大（减少不确定性程度最高）的问题来问你。其实 `ID3` 算法也是这么想的。 `ID3` 算法的思想是从训练集 `D` 中计算每个特征的信息增益，然后看哪个最大就选哪个作为当前结点。然后继续重复刚刚的步骤来构建决策树。

6.3 决策树算法流程

我们最终的目的是根据创建的决策树模型对测试集数据进行预测，算法实现流程如下：

- 1.计算训练样本信息增益
- 2.获得信息增益最高的特征
- 3.递归创建决策树
- 4.根据决策树模型对测试集数据进行预测

6.4 动手实现决策树

```
#encoding=utf8
import numpy as np

# 计算熵
def calcInfoEntropy(label):
    '''
    label(narray):样本标签
    '''
    label_set = set(label)
    result = 0
    for l in label_set:
        count = 0
        for j in range(len(label)):
            if label[j] == l:
                count += 1
        # 计算标签在数据集中出现的概率
        p = count / len(label)
        # 计算熵
        result -= p * np.log2(p)
    return result

#计算条件熵
def calcHDA(feature,label,index,value):
    '''
    input:
        feature(ndarray):样本特征
        label(ndarray):样本标签
        index(int):需要使用的特征列索引
        value(int):index所表示的特征列中需要考察的特征值
    output:
        HDA(float):信息熵
    '''
    count = 0
    # sub_feature和sub_label表示根据特征列和特征值分割出的子数据集中的特征和标签
    sub_feature = []
    sub_label = []
    for i in range(len(feature)):
        if feature[i][index] == value:
            count += 1
            sub_feature.append(feature[i])
            sub_label.append(label[i])
    pHA = count / len(feature)
    e = calcInfoEntropy(sub_label)
    HDA = pHA * e
    return HDA

#计算信息增益
def calcInfoGain(feature, label, index):
    '''
    input:
        feature(ndarry):测试用例中字典里的feature
        label(ndarray):测试用例中字典里的label
        index(int):测试用例中字典里的index，即feature部分特征列的索引。该索引指的是feature中第几个特征，如index:0表
示使用第一个特征来计算信息增益。
    output:
```

```
        InfoGain(float):信息增益
    '''
    base_e = calcInfoEntropy(label)
    f = np.array(feature)
    # 得到指定特征列的值的集合
    f_set = set(f[:, index])
    sum_HDA = 0
    # 计算条件熵
    for value in f_set:
        sum_HDA += calcHDA(feature, label, index, value)
    # 计算信息增益
    InfoGain = base_e - sum_HDA
    return InfoGain

# 获得信息增益最高的特征
def getBestFeature(feature, label):
    '''
    input:
        feature(ndarray):样本特征
        label(ndarray):样本标签
    output:
        best_feature(int):信息增益最高的特征
    '''
    max_infogain = 0
    best_feature = 0
    for i in range(len(feature[0])):
        infogain = calcInfoGain(feature, label, i)
        if infogain > max_infogain:
            max_infogain = infogain
            best_feature = i
    return best_feature

#创建决策树
def createTree(feature, label):
    '''
    input:
        feature(ndarray):训练样本特征
        label(ndarray):训练样本标签
    output:
        tree(dict):决策树模型
    '''
    # 样本里都是同一个label没必要继续分叉了
    if len(set(label)) == 1:
        return label[0]
    # 样本中只有一个特征或者所有样本的特征都一样的话就看哪个label的票数高
    if len(feature[0]) == 1 or len(np.unique(feature, axis=0)) == 1:
        vote = {}
        for l in label:
            if l in vote.keys():
                vote[l] += 1
            else:
                vote[l] = 1
        max_count = 0
        vote_label = None
        for k, v in vote.items():
            if v > max_count:
                max_count = v
                vote_label = k
        return vote_label
    # 根据信息增益拿到特征的索引
    best_feature = getBestFeature(feature, label)
```

```
    tree = {best_feature: {}}
    f = np.array(feature)
    # 拿到bestfeature的所有特征值
    f_set = set(f[:, best_feature])
    # 构建对应特征值的子样本集sub_feature, sub_label
    for v in f_set:
        sub_feature = []
        sub_label = []
        for i in range(len(feature)):
            if feature[i][best_feature] == v:
                sub_feature.append(feature[i])
                sub_label.append(label[i])
        # 递归构建决策树
        tree[best_feature][v] = createTree(sub_feature, sub_label)
    return tree

#决策树分类
def dt_clf(train_feature,train_label,test_feature):
    '''
    input:
        train_feature(ndarray):训练样本特征
        train_label(ndarray):训练样本标签
        test_feature(ndarray):测试样本特征
    output:
        predict(ndarray):测试样本预测标签
    '''
    #创建决策树
    tree = createTree(train_feature,train_label)
    result = []
    #根据tree与特征进行分类
    def classify(tree,test_feature):
        #如果tree是叶子节点，返回tree
        if not isinstance(tree,dict):
            return tree
        #根据特征值走入tree中的分支
        t_index,t_value = list(tree.items())[0]
        f_value = test_feature[t_index]
        #如果分支依然是tree
        if isinstance(t_value,dict):
            #根据tree与特征进行分类
            classLabel = classify(tree[t_index][f_value],test_feature)
            return classLabel
        else:
            #返回特征值
            return t_value
    for f in test_feature:
        result.append(classify(tree,f))
    predict = np.array(result)
    return predict
```

6.5 实战案例

鸢尾花数据

鸢尾花数据集是一类多重变量分析的数据集，一共有`150`个样本，通过花萼长度，花萼宽度，花瓣长度，花瓣宽度`4`个特征预测鸢尾花卉属于（`Setosa`，`Versicolour`，`Virginica`）三个种类中的哪一类。

数据集中部分数据如下所示：

花萼长度	花萼宽度	花瓣长度	花瓣宽度
5.1	3.5	1.4	0.2
4.9	3.2	1.4	0.2
4.7	3.1	1.3	0.2

其中每一行代表一个鸢尾花样本各个属性的值。

数据集中部分标签如下图所示：

标签
0
1
2

标签中的值`0`，`1`，`2`分别代表鸢尾花三种不同的类别。

我们可以直接使用`sklearn`直接对数据进行加载，代码如下：

```
from sklearn.datasets import load_iris
#加载鸢尾花数据集
iris = load_iris()
#获取数据特征与标签
x,y = iris.data.astype(int),iris.target
```

然后我们划分出训练集与测试集，训练集用来训练模型，测试集用来检测模型性能。代码如下：

```
from sklearn.model_selection import train_test_split
#划分训练集测试集，其中测试集样本数为整个数据集的20%
train_feature,test_feature,train_label,test_label = train_test_split(x,y,test_size=0.2,random_state=666)
```

进行分类

然后我们再使用实现的决策树分类方法就可以对测试集数据进行分类：

```
predict = dt_clf(train_feature,train_label,test_feature)
predict
>>>array([1, 2, 1, 2, 0, 1, 1, 2, 1, 1, 1, 0, 0, 0, 2, 1, 0, 2, 2, 2, 1, 0,2, 0, 1, 1, 0, 1, 2, 2])
```

再根据测试集标签，可以计算出正确率：

```
acc = np.mean(predict==test_label)
acc
>>>1.0
```

可以看到，使用决策树对鸢尾花进行分类，正确率可以达到 `100%`

第七章 **k-**均值

7.1 k-均值算法思想

`k-means` 是属于机器学习里面的非监督学习，通常是大家接触到的第一个聚类算法，其思想非常简单，是一种典型的基于距离的聚类算法。`k-means`（K-均值）聚类，之所以称为 K-均值 是因为它可以发现 `K` 个簇，且每个簇的中心采用簇中所含值的均值计算而成。簇内的样本连接紧密，而簇之间的距离尽量大。简单来讲，其思想就是物以类聚。

7.2 k-均值算法原理

假设我们有 `k` 个簇：$(c_1, c_2, ..., c_k)$

则我们的目的就是使的簇内的每个点到簇的质心的距离最小，即最小化平方误差 `MSE` ：

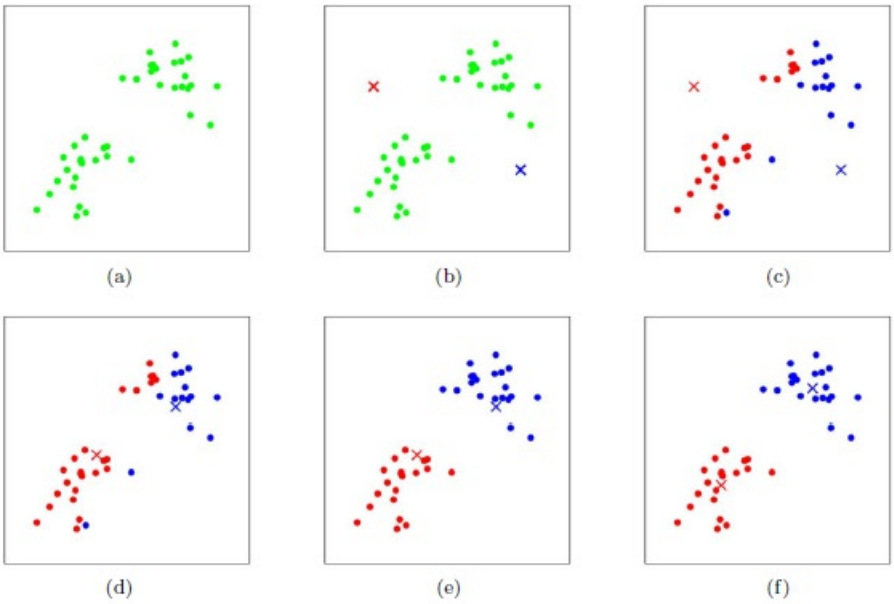
$$\sum_{i=1}^{k}\sum_{x\in c_i}(x-u_i)^2$$

其中，u_i为质心，表达式为：

$$\frac{1}{|c_i|}\sum_{x\in c_i}x$$

$|c_i|$表示集合内样本个数。

想要直接求得最小值是非常困难的，通常我们使用启发式的迭代方法，过程如下图：



- 图 `b` :假设 `k=2` ，我们最开始先随机初始 `2` 个质心(红色与蓝色的点)。
- 图 `c` :计算每个样本到两个质心的距离，并将其归为与其距离最近的质心那个簇。
- 图 `d` :更新质心，我们可以看到，红色与蓝色的点位置有了变化。
- 图 `e` :重新计算样本到质心距离，并重新划分样本属于哪个簇。
- 图 `f` :直到质心位置变换小于阈值或者达到迭代次数的最大值时停止迭代。

所以该算法的伪代码如下：

```
随机初始化k个质心
设置最大迭代次数
设置质心变化的最小阈值
while 当前迭代次数 < 最大迭代次数:
    计算每个样本分别到k个质心的距离
```

```
    对每个样本打上标记，标记为离哪个质心最近
    按照质心计算公式计算出k个新质心
    if 新质心与老质心的距离 < 质心变化的最小阈值:
        break
```

此时样本上的标记就代表了样本属于k个簇中的哪个簇，而k个质心表示k个簇的中心点

7.3 k-均值算法流程

`k-means` 算法流程如下：

- 1.随机初始 `k` 个点，作为类别中心。
- 2.对每个样本将其标记为距离类别中心最近的类别。
- 3.将每个类别的质心更新为新的类别中心。
- 4.重复步骤 `2、3` ，直到类别中心的变换小于阈值。

7.4 动手实现k-均值

```
#encoding=utf8
import numpy as np

# 计算一个样本与数据集中所有样本的欧氏距离的平方
def euclidean_distance(one_sample, X):
    '''
    input:
        one_sample(ndarray):单个样本
        X(ndarray):所有样本
    output:
        distances(ndarray):单个样本到所有样本的欧氏距离平方
    '''
    one_sample = one_sample.reshape(1, -1)
    distances = np.power(np.tile(one_sample, (X.shape[0], 1)) - X, 2).sum(axis=1)
    return distances

# 从所有样本中随机选取k个样本作为初始的聚类中心
def init_random_centroids(k,X):
    '''
    input:
        k(int):聚类簇的个数
        X(ndarray):所有样本
    output:
        centroids(ndarray):k个簇的聚类中心
    '''
    n_samples, n_features = np.shape(X)
    centroids = np.zeros((k, n_features))
    for i in range(k):
        centroid = X[np.random.choice(range(n_samples))]
        centroids[i] = centroid
    return centroids

# 返回距离该样本最近的一个中心索引[0, k)
def _closest_centroid(sample, centroids):
    '''
    input:
        sample(ndarray):单个样本
        centroids(ndarray):k个簇的聚类中心
    output:
        closest_i(int):最近中心的索引
    '''
    distances = euclidean_distance(sample, centroids)
    closest_i = np.argmin(distances)
    return closest_i

# 将所有样本进行归类，归类规则就是将该样本归类到与其最近的中心
def create_clusters(k,centroids, X):
    '''
    input:
        k(int):聚类簇的个数
        centroids(ndarray):k个簇的聚类中心
        X(ndarray):所有样本
    output:
        clusters(list):列表中有k个元素，每个元素保存相同簇的样本的索引
    '''
```

```
    clusters = [[] for _ in range(k)]
    for sample_i, sample in enumerate(X):
        centroid_i = _closest_centroid(sample, centroids)
        clusters[centroid_i].append(sample_i)
    return clusters

# 对中心进行更新
def update_centroids(k,clusters, X):
    '''
    input:
        k(int):聚类簇的个数
        X(ndarray):所有样本
    output:
        centroids(ndarray):k个簇的聚类中心
    '''
    n_features = np.shape(X)[1]
    centroids = np.zeros((k, n_features))
    for i, cluster in enumerate(clusters):
        centroid = np.mean(X[cluster], axis=0)
        centroids[i] = centroid
    return centroids

# 将所有样本进行归类，其所在的类别的索引就是其类别标签
def get_cluster_labels(clusters, X):
    '''
    input:
        clusters(list):列表中有k个元素，每个元素保存相同簇的样本的索引
        X(ndarray):所有样本
    output:
        y_pred(ndarray):所有样本的类别标签
    '''
    y_pred = np.zeros(np.shape(X)[0])
    for cluster_i, cluster in enumerate(clusters):
        for sample_i in cluster:
            y_pred[sample_i] = cluster_i
    return y_pred

# 对整个数据集X进行Kmeans聚类，返回其聚类的标签
def predict(k,X,max_iterations,varepsilon):
    '''
    input:
        k(int):聚类簇的个数
        X(ndarray):所有样本
        max_iterations(int):最大训练轮数
        varepsilon(float):最小误差阈值
    output:
        y_pred(ndarray):所有样本的类别标签
    '''
    # 从所有样本中随机选取k样本作为初始的聚类中心
    centroids = init_random_centroids(k,X)
    # 迭代，直到算法收敛(上一次的聚类中心和这一次的聚类中心几乎重合)或者达到最大迭代次数
    for _ in range(max_iterations):
        # 将所有进行归类，归类规则就是将该样本归类到与其最近的中心
        clusters = create_clusters(k,centroids, X)
        former_centroids = centroids
        # 计算新的聚类中心
        centroids = update_centroids(k,clusters, X)
        # 如果聚类中心几乎没有变化，说明算法已经收敛，退出迭代
        diff = centroids - former_centroids
        if diff.any() < varepsilon:
            break
```

```
    y_pred = get_cluster_labels(clusters, X)
    return y_pred
```

7.5 实战案例

鸢尾花数据

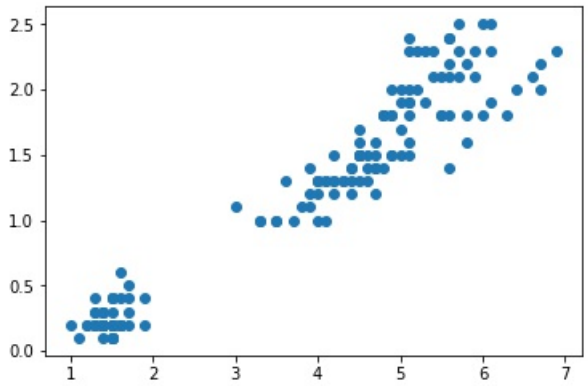
本次我们使用的仍然是鸢尾花数据，不过为了能够进行可视化我们只使用数据中的两个特征：

```
from sklearn.datasets import load_iris

iris = load_iris()
x,y = iris.data,iris.target
x = x[:,2:]
```

可视化数据分布：

```
import matplotlib.pyplot as plt

plt.scatter(x[:,0],x[:,1])
plt.show()
```

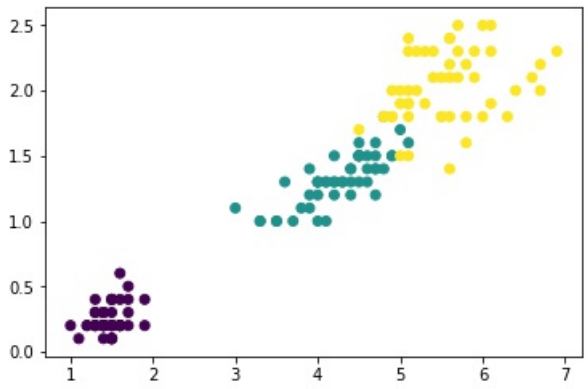
可视化结果：



我们可以先根据数据的真实标签查看数据类别情况：

```
import matplotlib.pyplot as plt

plt.scatter(x[:,0],x[:,1],c=y)
plt.show()
```

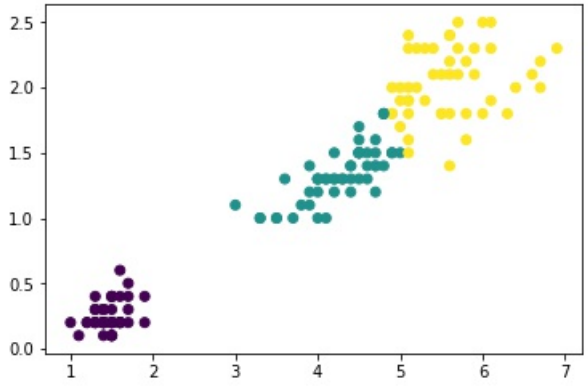
效果如下：



进行聚类

最后，使用我们实现的 `k-means` 方法对数据进行聚类并查看聚类效果：

```
predict = predict(3,x,500,0.0001)

plt.scatter(x[:,0],x[:,1],c=predict)
plt.show()
```



可以发现，使用实现的方法进行聚类的结果与真实情况非常吻合。

第八章 Apriori算法

8.1 关联规则与Apriori算法

什么是关联规则

顾名思义，关联规则就是发现数据背后存在的某种规则或者联系。举个例子：通过调研超市顾客购买的东西，可以发现 30% 的顾客会同时购买薯片和可乐，而在购买薯片的顾客中有 80% 的人购买了可乐，这就存在一种隐含的关系：`薯片->可乐`，也就是说购买薯片的顾客会有很大可能购买可乐，因此商场可以将薯片和可乐放在同一个购物区，方便顾客购买。这样一来，很有可能会在无形之中提高超市的销售业绩。

怎样挖掘关联规则

想要从海量数据中挖掘出关联规则是一件比较麻烦的事情，因为主要问题在于，寻找物品的不同组合是一项十分耗时的任务，所需的计算代价很高，蛮力搜索方法并不能解决这个问题，所以需要用更智能的方法在合理的时间范围内找到频繁项集。而 Apriori 算法能够帮助我们从频繁项集中挖掘出关联规则。

频繁项集与关联规则

刚刚提到了一个新名词：频繁项集，频繁项集表示的是经常出现在一起的物品的集合。而关联规则指的是两个物品之间可能存在很强的某种关系。

假设现在这样的一份数据：

票号	商品
233	薯片 西瓜
234	抹布 可乐 纸巾 电池
235	薯片 可乐 纸巾 果汁
236	西瓜 薯片 可乐 纸巾
237	西瓜 薯片 可乐 果汁

现在想要找到频繁项集，但是频繁应该怎样来定义或者量化呢？通常来说，会根据项集的支持度和可信度来衡量项集是否频繁。

一个项集的支持度表示的是数据集中包含该项集的记录所占的比例。从上表中可知，`{薯片}` 的支持度为 4/5 。`{薯片，可乐}` 的支持度为 3/5 。很明显，支持度这个指标是针对于项集的。

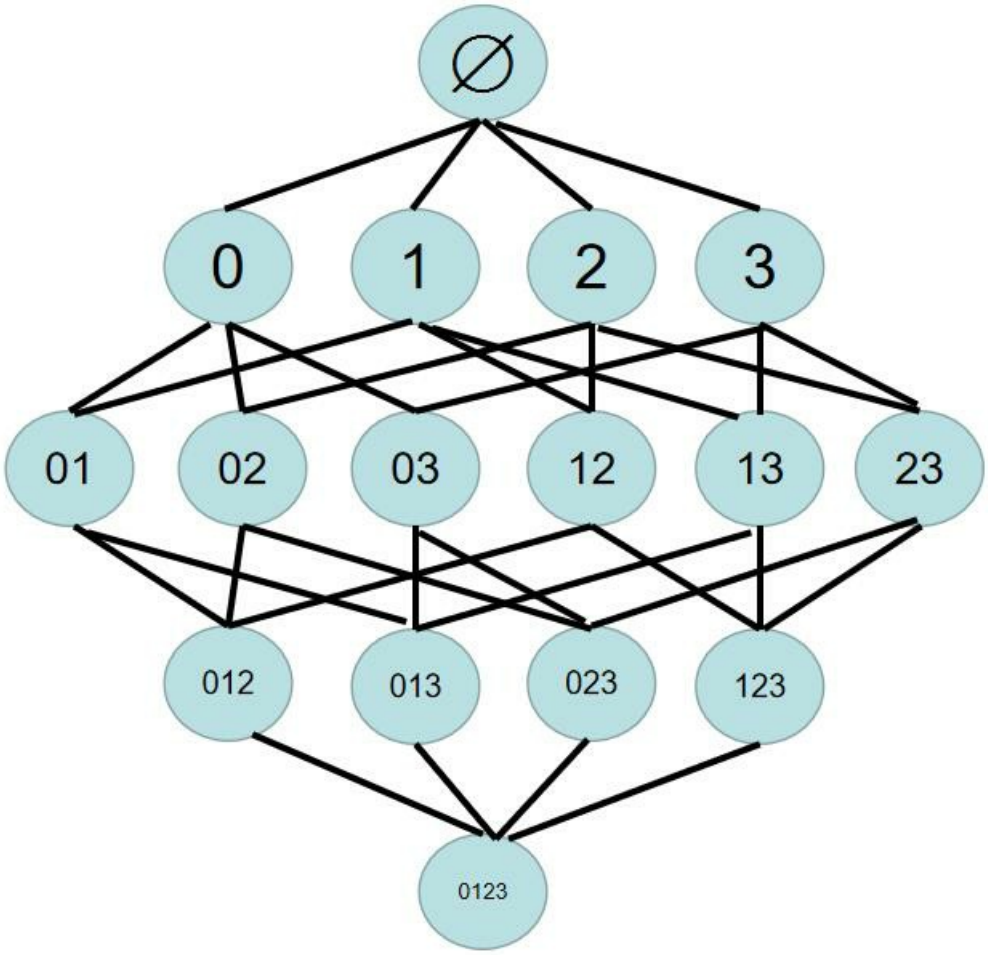
而可信度是针对关联规则而言的。例如 `{可乐}->{纸巾}` 的可信度为 `支持度({可乐，纸巾})/支持度({可乐})` ,即 `{可乐}->{纸巾}` 的可信度为 0.75 。这个 0.75 意味着对于包含可乐的所有记录，这个关联规则对其中 75% 的记录都适用。

支持度和可信度是用来量化关联分析是否成功的方法。假设想找到支持度大于 0.8 的所有项集 ，应该如何去做？一个办法是生成一个物品所有可能组合的清单，然后对每一种组合统计它出现的频繁程度，但当物品成千上万时，上述做法非常非常慢。此时就可以使用 Apriori 算法来寻找频繁项集，并从频繁项集中挖掘出关联规则。

8.2 Apriori算法原理

Apriori原理

假设我们在经营一家商品种类并不多的小卖铺，我们对那些经常在一起被购买的商品非常感兴趣。我们只有 4 种商品：薯片，可乐，纸巾和电池。那么所有可能被一起购买的商品组合都有哪些？这些商品组合可能只有一种商品，比如薯片, 也可能包括两种、三种或者所有四种商品。我们并不关心某人买了两包薯片以及四对电池的情况，我们只关心他购买了一种或多种商品。所以这四种商品的组合图如下(其中用 0 表示薯片，1 表示可乐，2 表示纸巾， 3 表示电视)：

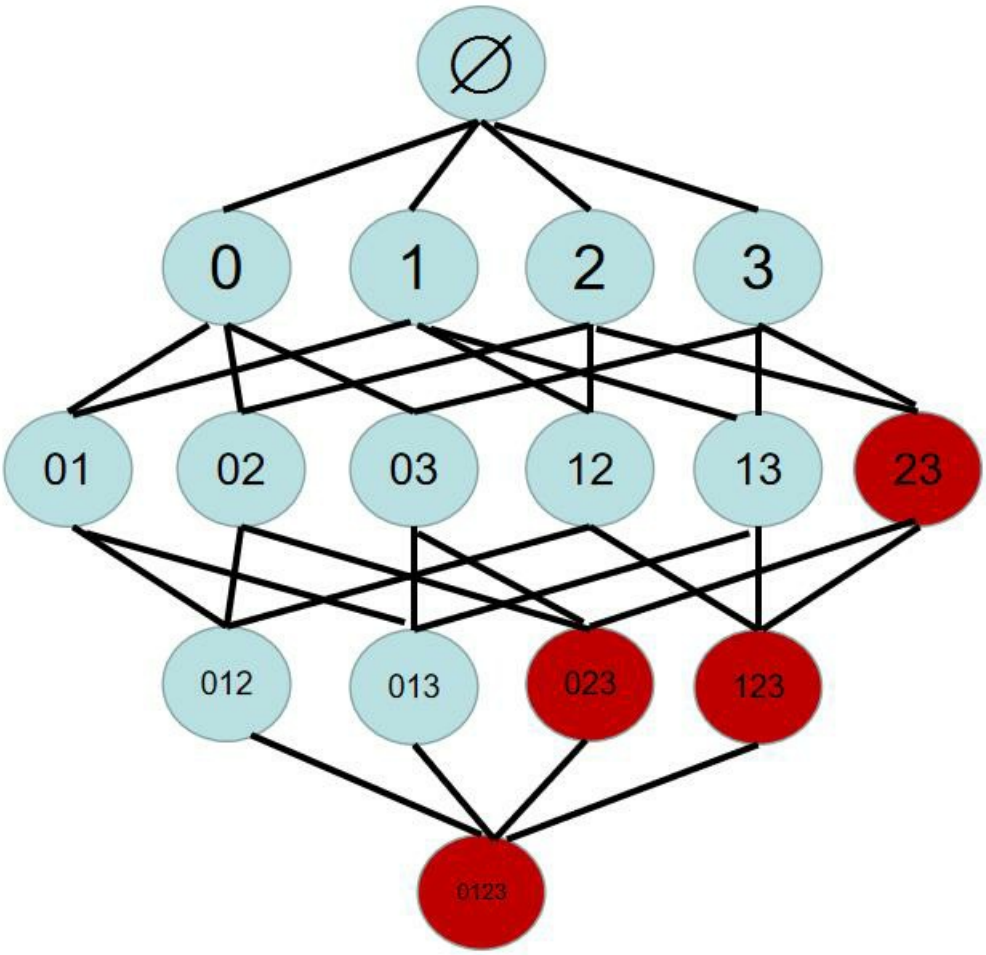


上一节说过，我们的目标是找到经常在一起购买的物品集合，并且我们通常会使用集合的支持度来度量其出现的频率。一个集合的支持度是指有多少比例的交易记录包含该集合。如何对一个给定的集合，比如 `{0, 3}` ，来计算其支持度？很显然，可以遍历每条记录并检查该记录包含 0 和 3 ，如果记录确实同时包含这两项，那么就增加总计数值。在扫描完所有数据之后，使用统计得到的总数除以总的交易记录数，就可以得到支持度。这个计算过程还仅仅是对 `{0, 3}` 这个项集而言，如果想要将海量数据中所有可能出现的项集的支持度给算出来的话，那么很用可能算一辈子也算不完。

为了降低所需的计算时间，研究人员发现一种所谓的 Apriori 原理。Apriori 原理可以帮我们减少可能感兴趣的项集。Apriori 原理是说如果某个项集是频繁的，那么它的所有子集也是频繁的。

对于我们开小卖铺的例子，如果 `{0, 1}` 这个项集是频繁的，那么 `{0}` 和 `{1}` 也一定是频繁的。如果将这个命题进行逆否处理，那么会得到另一个结论，就是如果一个项集是非频繁的，那么它的所有超集都是非频繁的。

那这个结论有什么用呢？假设 `{2, 3}` 这个项集是非频繁的，那么跟它有关的 `{0, 2, 3}` ， `{1, 2, 3}` ， `{0, 1, 2, 3}` 都是非频繁的。也就是说算完 `{2, 3}` 的支持度发现不是频繁的，那么后面 3 个项集的支持度就不需要算了，这样就减小了计算的时间复杂度。



Apriori算法流程

Apriori 算法的两个输人参数分别是最小支持度和数据集。该算法首先会生成所有单个物品的项集列表。接着扫描交易记录来查看哪些项集满足最小支持度要求，那些不满足最小支持度的集合会被去掉。然后，对剩下来的集合进行组合以生成包含两个元素的项集。接下来，再重新扫描交易记录，去掉不满足最小支持度的项集。该过程重复进行直到所有项集都被去掉。

所以 Apriori 算法的伪代码如下：

```
while 集合中的项的个数 > 0:
    构建一个由 k 个项组成的候选项集的列表
    确认每个项集都是频繁的
    保留频繁项集并构建 k+1 项组成的候选项集的列表
```

从频繁项集中挖掘关联规则

要找到关联规则，需要从一个频繁项集开始。我们知道集合中的元素是不重复的，但我们想知道基于这些元素能否获得其他内容。例如某个元素或者某个元素集合可能会推导出另一个元素 。从小卖铺的例子可以得到，如果有一个频繁项集 `{薯片，西瓜}` ，那么就可能有一条关联规 则 `薯片->西瓜` 。这意味着如果有人购买了薯片，那么在统计上他会购买西瓜的概率较大。

但是，这一条反过来并不总是成立。也就是说，即使 `薯片->西瓜` 统计上显著，那么 `薯片->西瓜` 也不一定成立。(从逻辑研究上来讲，箭头左边的集合称作前件，箭头右边的集合称为后件。)

那么怎样挖掘关联规则呢？在发现频繁项集时我们发现的是高于最小支持度的频繁项集，对于关联规则，也是用这种类似的方法。以小卖铺的例子为例，从项集 `{0, 1, 2, 3}` 产生的关联规则中，找出可信度高于最小可信度的关联规则。(**PS:Apriori** 原理对于关联规则同样适用。)

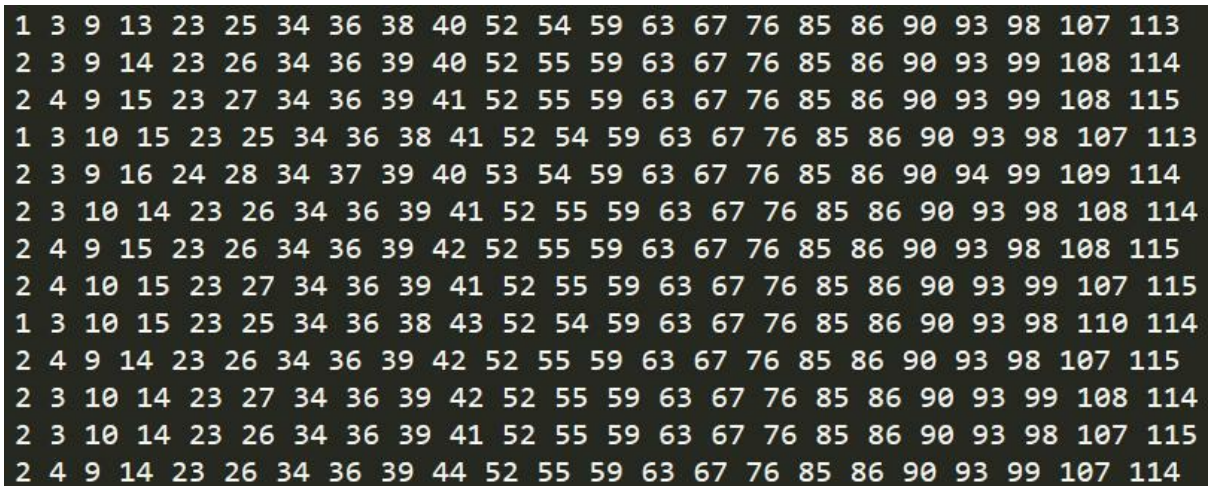
8.3 动手实现Apriori

```
# 寻找频繁项集部分 Begin

# 构建只有一个元素的项集， 假设dataSet为[[1, 2], [0, 1]. [3, 4]]
# 那么该项集为frozenset({0}), frozenset({1}), frozenset({2}),frozenset({3}), frozenset({4})
def createC1(dataSet):
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    return map(frozenset, C1)

# 从只有k个元素的项集，生成有k+1个元素的频繁项集，排除掉支持度小于最小支持度的项集
# D为数据集，ck为createC1的输出，minsupport为最小支持度
def scanD(D, ck, minsupport):
    ssCnt = {}
    for tid in D:
        for can in ck:
            if can.issubset(tid):
                if not ssCnt.has_key(can):
                    ssCnt[can]=1
                else:
                    ssCnt[can] +=1
    numItems = float(len(D))
    reList = []
    supportData = {}
    for key in ssCnt:
        support = ssCnt[key]/numItems
        if support >= minsupoort:
            reList.insert(0, key)
        supportData[key] = support
    #reList为有k+1个元素的频繁项集，supportData为频繁项集对应的支持度
    return reList, supportData

#构建含有k个元素的频繁项集
#如输入为{0},{1},{2}会生成{0,1},{0, 2},{1,2}
def aprioriGen(Lk, k):
    retList = []
    lenLk = len(LK)
    for i in range(lenLk):
        for j in range(i+1, lenKk):
            L1 = list(Lk[i])[k:-2]
            L2 = list(Lk[j])[:k-2]
            if L1 == L2:
                relist.append(Lk[i] | Lk[j])
    return reList

#生成候选的频繁项集列表，以及候选频繁项集的支持度，因为在算可信度时要用到
def apriori(dataSet, minsupport=0.5):
    C1 = creatC1(dataSet)
    D = map(set, dataSet)
    L1, supportData = scanD(dataSet, C1, minsupport)
    L = [L1]
    k = 2
```

```
    while (len(L[k-2])>0):
        Ck = aprioriGen(L[k-2], k)
        Lk, supK = scanD(D, Ck, minsupport)
        supportData.update(supK)
        L.append(Lk)
        k += 1
    return L, supportData

# 寻找频繁项集部分 End

# 挖掘关联规则部分 Begin

# 计算关联规则的可信度，并排除可信度小于最小可信度的关联规则
# freqSet为频繁项集，H为规则右边可能出现的元素的集合，supportData为频繁项集的支持度，brl为存放关联规则的列表，minCon
f为最小可信度
def calcConf(freqSet, H, supportData, brl, minConf = 0.7):
    prunedH = []
    for conseq in H:
        conf = supportData[freqSet]/supportData[freqSet - conseq]
    if conf >= minConf:
        brl.append((freqSet - conseq, conseq, conf))
        prunedH.append(conseq)
    return prunedH

# 从频繁项集中生成关联规则
# freqSet为频繁项集，H为规则右边可能出现的元素的集合，supportData为频繁项集的支持度，brl为存放关联规则的列表，minCon
f为最小可信度
def ruleFromConseq(freqSet, H, supportData, brl, minConf = 0.7):
    m = len(H[0])
    if len(freqSet) > m+1:
        Hmp1 = aprioriGen(H, m+1)
        Hmp1 = calcConf(freqSet, Hmp1, supporData, brl, minConf)
        if len(Hmp1) > 1:
            ruleFromConseq(freqSet, Hmp1, supportData, brl, minConf)

# 从频繁项集中挖掘关联规则
# L为频繁项集， supportData为频繁项集的支持度，minConf为最小可信度
def generateRules(L, supportData, minConf = 0.7):
    digRuleList = []
    for i in range(1, len(L)):
        # freqSet为含有i个元素的频繁项集
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            if i > 1:
                # H1为关联规则右边的元素的集合
                rulesFromConseq(freqSet, H1, supportData, digRuleList, minConf)
            else:
                calcConf(freqSet, H1, supportData, digRuleList, minConf)
    return digRuleList

# 挖掘关联规则部分 End
```

8.4 实战案例

最后我们来尝试使用 Apriori 算法来寻找毒蘑菇中的一些公共特征，利用这些特征就能避免吃到那些有毒的蘑菇。现在有这样一个数据集，数据集合中有一个关于蘑菇的 23 种特征的数据集，每一个特征都包含一 个标称数据值。部分数据截图如下：



其中每行的第一列为标签，1 表示蘑菇无毒， 2 表示蘑菇有毒。接下来，为了找到毒蘑菇的共性特征，我们可以使用刚刚动手实现的 Apriori 算法来含有标签值为 2 的频繁项集。如寻找含有 3 个元素的频繁项集。

```
import pandas as pd

# 读取蘑菇数据
data = pd.read_csv('./data.csv').values()

# 调用上一节中实现的apriori算法
L, _ = apriori(data, minsupport=0.4)

# 遍历含有3个元素的频繁项集
for item in L[3]:
    if item.intersection(2):
        # 打印出含有标签值为2的频繁项集
        print(item)
```

结果如下：

```
frozenset([63, 59, 2, 93])
frozenset([39, 2, 53, 34])
frozenset([2, 59, 23, 85])
frozenset([2, 59, 90, 85])
frozenset([39, 2, 36, 34])
frozenset([39, 63, 2, 85])
```

可以看出，当这些特征一起出现时，表示该蘑菇很有可能是有毒的。

第九章 PageRank

9.1 什么是PageRank

PageRank的Page可是认为是网页，表示网页排名，也可以认为是Larry Page(google 产品经理)，因为他是这个算法的发明者之一，还是google CEO。

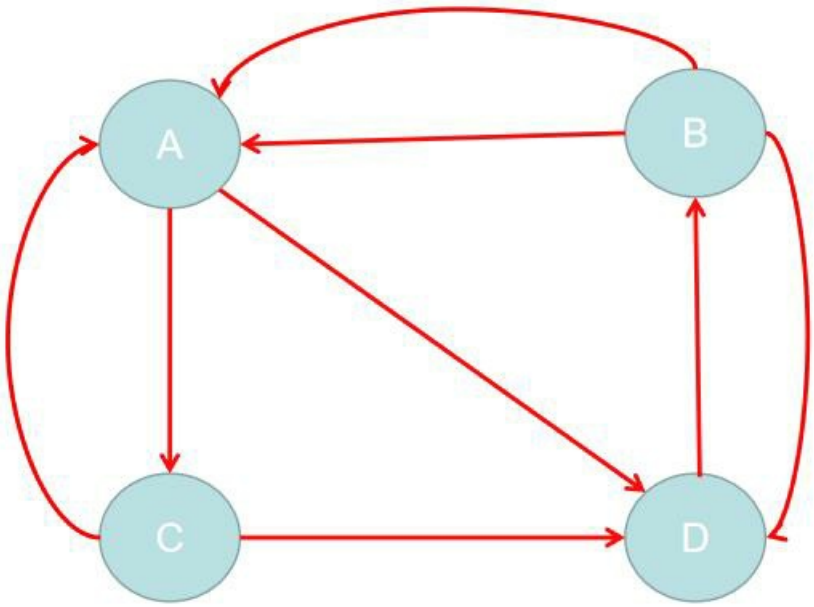
PageRank算法计算每一个网页的PageRank值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank就是估计这个悠闲的上网者分布在各个网页上的概率。

当计算出上网者分布在各个网页上的概率之后，就可以将概率高的网页推荐给上网者以提高用户体验，同时也节省上网者查找资料的时间。

9.2 PageRank算法原理

最简单的场景

互联网中的网页可以看出是一个有向图，其中网页是结点，如果网页A有链接到网页B，则存在一条有向边 `A->B` ，下面是一个简单的示例：



这个例子中只有四个网页，如果当前在 A 网页，那么悠闲的上网者将会各以 1/3 的概率跳转到 B、C、D，这里的 3 表示 A 有 3 条出路，如果一个网页有 k 条出路，那么跳转任意一个出路上的概率是 1/k，同理 D 到 B、C 的概率各为 1/2 ，而 B 到 C 的概率为 0。一般用转移矩阵表示上网者的跳转概率，如果用 n 表示网页的数目，则转移矩阵 M 是一个 n 阶的方阵；如果网页 j 有 k 个出路，那么对每一个出链指向的网页 i，有 M[i][j]=1/k，而其他网页的 M[i][j]=0。上面示例图对应的转移矩阵如下：

$$M=\begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

一开始，假设上网者在每一个网页的概率都是相等的，即 1/n ，于是初始的概率分布就是一个所有值都为 1/n 的 n 维列向量 V0，用 V0 去右乘转移矩阵 M ，就得到了第一步之后上网者的概率分布向量 V1, 下面是 V1 的计算过程：

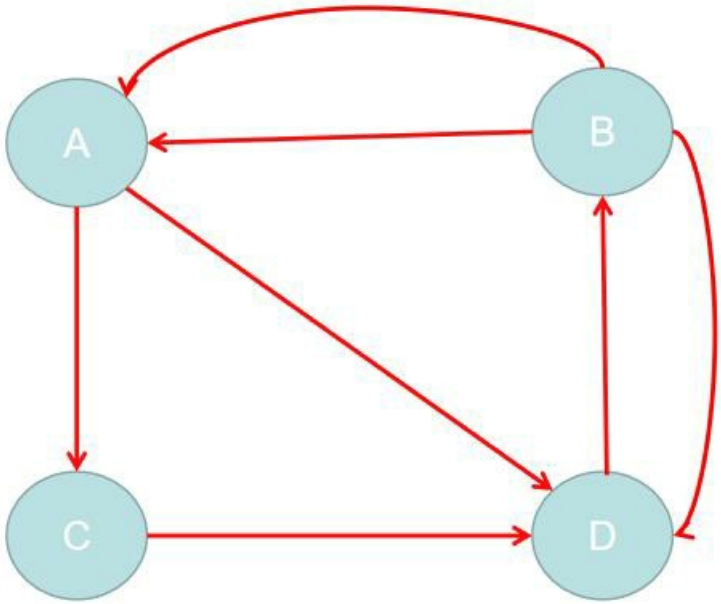
$$V_1 = MV_0 = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

得到了 V1 后，再用 V1 去右乘 M 得到 V2 ，一直下去，最终 V 会收敛，即 Vn=MV(n-1)，不断的迭代，最终 V=[3/9,2/9,2/9,2/9]。所以如果想要向该上网者推荐一个网页，那么就会推荐网页 A。因为它的概率最高。

更复杂的场景

上述上网者的行为是一个马尔科夫过程，要满足收敛性，需要具备一个条件：图是强连通的，即从任意网页可以到达其他任意网页。

然而，互联网上的网页不满足强连通的特性，因为有一些网页不指向任何网页，如果按照上面的计算，上网者到达这样的网页后便走投无路、四顾茫然，导致前面累计得到的转移概率被清零，这样下去，最终的得到的概率分布向量所有元素几乎都为 0 。假设我们把上面图中 C 到 A 的链接丢掉， C 变成了一个终止点。

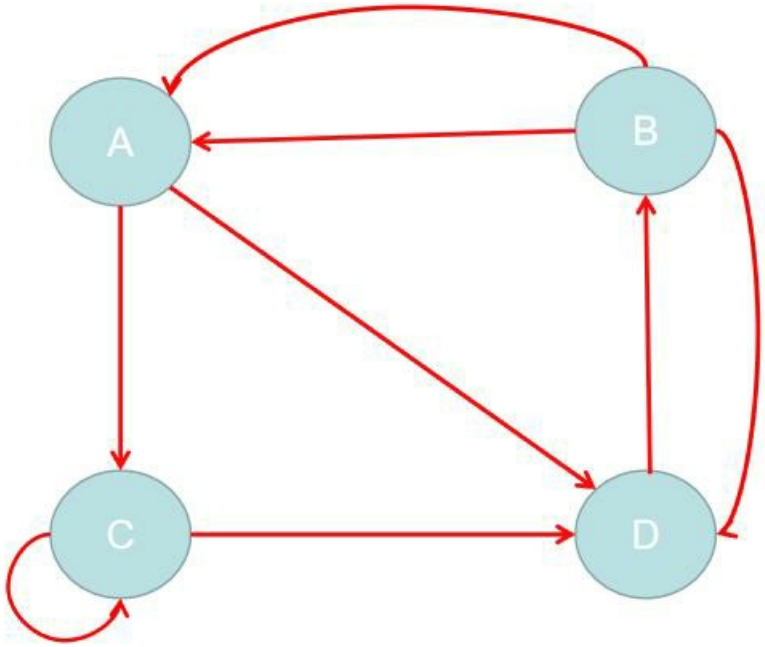


对应的转移矩阵 M 为：

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

这样一个转移矩阵一直迭代下去的话会发现 V=[0,0,0,0]。

当然，还有更加复杂的场景，例如有些网页不存在指向其他网页的链接，但存在指向自己的链接。如下图所示：



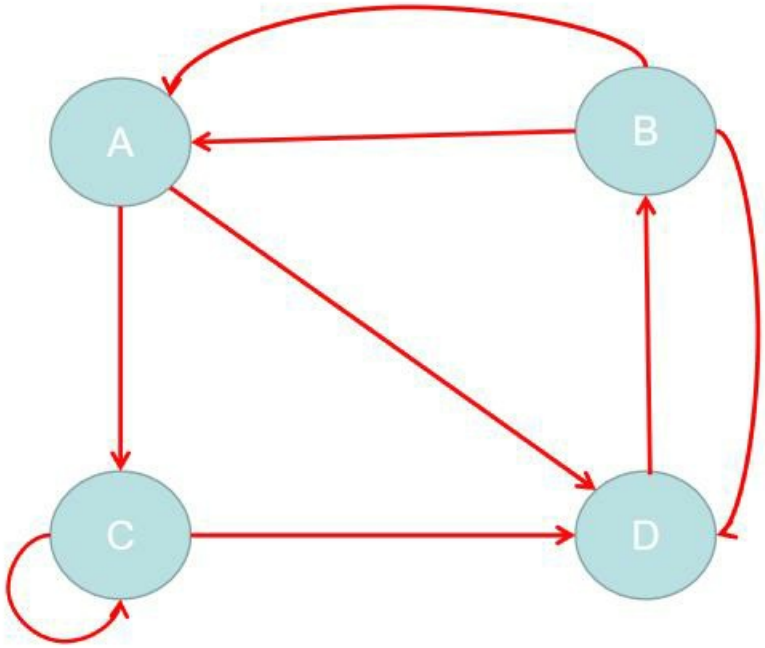
上网者跑到 C 网页后，就像陷入了漩涡，再也不能从 C 中出来，将最终导致概率分布值全部转移到 C 上来，这使得其他网页的概率分布值为 0 ，从而整个网页排名就失去了意义。

怎样解决复杂场景的问题

上面过程，我们忽略了一个问题，那就是上网者是一个悠闲的上网者，而不是一个愚蠢的上网者，我们的上网者是聪明而悠闲，他悠闲，漫无目的，总是随机的选择网页，他聪明，在走到一个终结网页或者一个陷阱网页（比如图中的 C ），不会傻傻的干着急，他会在浏览器的地址随机输入一个地址，当然这个地址可能又是原来的网页，但这里给了他一个逃离的机会，让他离开这万丈深渊。模拟聪明而又悠闲的上网者，对算法进行改进，每一步，上网者可能都不想看当前网页了，不看当前网页也就不会点击上面的连接，而上悄悄地在地址栏输入另外一个地址，而在地址栏输入而跳转到各个网页的概率是 1/n 。假设上网者每一步查看当前网页的概率为 a ，那么他从浏览器地址栏跳转的概率为 (1-a) ，于是迭代公式转化为：

$$V' = \alpha MV + (1-\alpha)V$$

假设场景如下图所示：



很显然，该图所对应的转移矩阵 M 为如下所示：

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

那么假设上网者查看当前网页的概率为 0.8 ，则根据公式可知：

$$V1 = \alpha MV + (1-\alpha)V = 0.8\begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} + 0.2\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix}$$

然后一直迭代下去，当收敛时可知 V=[3/9,2/9,2/9,2/9]。

9.3 动手实现PageRank

理解了PageRank算法原理之后，想要动手实现PageRank算法其实不难。代码如下：

```
from numpy import *

# 构造转移矩阵，其中a为有向图的邻接矩阵
def graphMove(a):
    b = transpose(a)  # b为a的转置矩阵
    c = zeros((a.shape), dtype=float)
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            c[i][j] = a[i][j] / (b[j].sum())  # 完成初始化分配
    return c


# 初始化V0
def firstPr(c):
    pr = zeros((c.shape[0], 1), dtype=float)
    for i in range(c.shape[0]):
        pr[i] = float(1) / c.shape[0]
    return pr


# 计算pageRank值
def pageRank(p, m, v):
    # 判断pr矩阵是否收敛,(v == p*dot(m,v) + (1-p)*v).all()判断前后的pr矩阵是否相等，若相等则停止循环
    while ((v == p * dot(m, v) + (
        1 - p) * v).all() == False):
        v = p * dot(m, v) + (1 - p) * v
    return v


if __name__ == "__main__":
    # 网页的邻接矩阵
    a = array([[0, 1, 1, 0],
            [1, 0, 0, 1],
            [1, 0, 0, 1],
            [1, 1, 0, 0]], dtype=float)
    M = graphMove(a)
    pr = firstPr(M)
    # 上网者查看当前网页的概率
    p = 0.8
    # 计算V
    print(pageRank(p, M, pr))
```

第十章 推荐系统

10.1 推荐系统概述

什么是推荐系统

你可能会有这样的经历，当你在电商网站闲逛时电商网站会根据你的历史行为轨迹分析出你的喜好，像你推荐一些你可能喜欢的商品。当你刷抖音，快手等小视频APP时，这些APP会根据你观看视频的时长、视频类型等数据进行判别，判别出你接下来可能更想要看到的视频。其实，这些基本上都是推荐系统的应用。

总结下来，使用的场景不外乎两个：帮你做内容推荐，或者说是筛选；另外就是做消费刺激，找到你想要买的东西。再抽象一点，其实考量的就是信息的相关行或者关联程度的问题。

基本概念

在开始前，我们先来明确一个问题，即什么是个性化推荐。简单来说，就是根据每个使用者的偏好，呈现出不同的内容。有一个误区要说明，所谓的热门推荐有时候网站上热卖或者流量最多的东西，并不一定是基于你的喜好分析的结果。

在明白问题界限之后，我们继续。接下来想想，既然是个性化的推荐，那么系统必然要维护一套用户模型，这个模型用来记录用户的偏好，从而预测这个用户可能感兴趣的内容。

那么用户模型的建立需要什么信息呢，一般来说可以从两种方式获取用户偏好信息。常见的一种方式就是由系统显示的询问用户，比如说在Apple music第一次使用时，它会让你选择自己喜欢的音乐类型和音乐人等信息；第二种方式就是隐式的记录用户使用时产生的数据。还是以Apple Music为例，假如你选择了摇滚和粤语的音乐类型，但是大部分时间你听的是周杰伦、Oasis的歌曲，那么系统会调整给你推荐的音乐，可能就不会听到陈奕迅的歌曲了。

除了用户本身的信息，那还有没其他可以利用的信息来进行推荐呢。答案是肯定的，其他用户产生的大量的信息或者物品本身的描述甚至与物品相关的因果知识，这些都是可以被使用的。

常见的推荐系统

不同的信息对应不同的推荐系统

- 协同过滤系统：前提就是如果另一个用户跟你曾经的购买经历很相似，那么当ta买了一本书而你还没有买的时候，那么你很有可能也会喜欢这本书，反过来也是一样的。这就相当于你跟这个用户隐式进行相互协作，所以成为协同过滤（Collaborative Filtering）。当然肯定不会只有你跟另一个用户两个人，这是会从大量用户集合中进行过滤分析的。
- 基于内容的推荐系统：无论是为了让你买东西还是给你找可能想阅读的文章，如果我们从这些被推荐项着手，试着分析它们的特征，比如说体裁、类型、风格、颜色等与你之前购买或者阅读的偏好信息进行匹配，这就是基于内容的推荐系统（Content-Based Recommendation）
- 基于知识的推荐系统。通过上面的介绍，我们可以知道大量多次的购买是CBS跟CF的前提，但是如果面对的是大量的单词购买者呢（比如说汽车），这个时候怎么办？我们可以考量使用更为精细和结构化的信息，比如说专业的特征等来构建明确的约束条件，同时，我们也看到，约束条件往往会跟用户有一

个交互式的引导，这样才能比较好的摸索出用户的喜好。

本章中主要介绍协同过滤算法，并讲解如何实现电影推荐功能。

10.2 基于矩阵分解的协同过滤算法思想

在推荐系统中，我们经常看到如下图的表格，表格中的数字代表用户对某个物品的评分，`0`代表未评分。我们希望能够预测目标用户对物品的评分，进而根据评分高低，将分高的物品推荐给用户。

y	物品1	物品2	物品3	物品4	物品5
用户1	5	5	0	1	1
用户2	5	0	4	1	1
用户3	1	0	1	5	5
用户4	1	1	0	4	0

基于矩阵分解的协同过滤算法正好能解决这个问题。

基于矩阵分解的协同过滤算法通常都会构造如下图所示评分表`y`，这里我们以电影为例：

y	电影1	电影2	电影3	电影4	电影5
用户1	5	5	0	1	1
用户2	5	0	4	1	1
用户3	1	0	1	5	5
用户4	1	1	0	4	0

我们认为，有很多因素会影响到用户给电影评分，如电影内容：感情戏，恐怖元素，动作成分，推理悬疑等等。假设我们现在想预测用户`2`对电影`2`的评分，用户`2`他很喜欢看动作片与推理悬疑，不喜欢看感情戏与恐怖的元素，而电影2只有少量的感情戏与恐怖元素，大部分都是动作与推理的剧情，则用户`2`对电影`2`评分可能很高，比如`5`分。

基于上面的设想，我们只要知道所有用户对电影内容各种元素喜欢程度与所有电影内容的成分，就能预测出所有用户对所有电影的评分了。 若只考虑两种元素则用户喜好表与电影内容表如下：

用户喜好表`x`：

x	因素1	因素2
用户1	5	0
用户2	5	0
用户3	0	5
用户4	0	5

值越大代表用户越喜欢某种元素。

电影内容表：`w`:

w	电影1	电影2	电影3	电影4	电影5
因素1	0.9	1.0	0.99	0.1	0
因素2	0	0.01	0	1.0	0.9

值越大代表电影中某元素内容越多。

用户 `2` 对电影 `2` 评分为：$5 \times 1.0 + 0 \times 0.01 = 5.0$

对于所有用户，我们可以将矩阵 `x` 与矩阵 `w` 相乘，得到所有用户对所有电影的预测评分如下表：

xw	电影**1**	电影**2**	电影**3**	电影**4**	电影**5**
用户1	4.5	5.0	4.95	0.5	0
用户2	4.5	5.0	4.95	0.5	0
用户3	0	0.05	0	5	4.5
用户4	0	0.05	0	5	4.5

假设电影评分表 `y` （为 `m` 行 `n` 列的矩阵）,我们考虑 `d` 种元素，则电影评分表可以分解为用户喜好表 `x` （为 `m` 行 `d` 列的矩阵），与电影内容表 `w` （为 `d` 行 `n` 列的矩阵）。其中 `d` 为超参数，大小由我们自己定。

基于矩阵分解的协同过滤算法思想为：一个用户评分矩阵可以分解为一个用户喜好矩阵与内容矩阵，我们只要能找出正确的用户喜好矩阵参数与内容矩阵参数（即表内的值），就能对用户评分进行预测，再根据预测结果对用户进行推荐。

所以不难看出，基于矩阵分解的协同过滤算法的流程如下：

- 1.随机初始矩阵值
- 2.构造损失函数，求得矩阵参数梯度
- 3.进行梯度下降，更新矩阵参数值
- 4.喜好矩阵与内容矩阵相乘得到预测评分
- 5.根据预测评分进行推荐

10.3 基于矩阵分解的协同过滤算法原理

将用户喜好矩阵与内容矩阵进行矩阵乘法就能得到用户对物品的预测结果，而我们的目的是预测结果与真实情况越接近越好。所以，我们将预测值与评分表中已评分部分的值构造平方差损失函数：

$$loss = \frac{1}{2} \sum_{(i,j) \in r(i,j)=1} (\sum_{l=1}^{d} x_{il} w_{lj} - y_{ij})^2$$

其中：

- i:第i个用户
- j:第j个物品
- d:第d种因素
- x:用户喜好矩阵
- w:内容矩阵
- y:评分矩阵
- r:评分记录矩阵，无评分记为0，有评分记为1。r(i,j)=1代表用户i对物品j进行过评分，r(i,j)=0代表用户i对物品j未进行过评分

损失函数 `python` 实现代码如下：

```
import numpy as np
loss = np.mean(np.multiply((y-np.dot(x,w))**2,record))
```

其中， `record` 为评分记录矩阵。

我们的目的就是最小化平方差损失函数，通常机器学习都是使用梯度下降的方法来最小化损失函数得到正确的参数。

梯度下降可以看成是我们被蒙住了双眼，然后要从山顶走到山下的过程。既然是被蒙住双眼，那么只能用脚去小步试探，看看哪个方向是朝着山下最抖的方向，找到了这个方向后就往该方向走一小步。然后不断地试探，走，试探，走，最终就可能会成功地走到山下。

如果我们将损失函数的值的大小看成是山的高度，那么我们就可以使用这种类似爬山的梯度下降算法来求解。其中，试探这个动作就是计算参数对于损失函数的偏导(即梯度)，走这个动作就是根据梯度来更新参数，至于走的时候步子迈多大就是学习率。

因此梯度下降算法的伪代码如下：

```
设置学习率 α
设置损失值变化的最小阈值 β
while 到达迭代次数:
    计算当前参数 θ 的损失值 J
    计算参数对损失函数的偏导 G
    计算新参数 Newθ，公式为 Newθ = θ - αG
    计算参数为 Newθ 时的损失值 NewJ
    if abs(NewJ - j) < β:
        break
```

对每个参数求得偏导如下：

$$\frac{\partial loss}{\partial x_{ik}} = \sum_{j \in r(i,j)=1} (\sum_{l=1}^{d} x_{il} w_{lj} - y_{ij}) w_{kj}$$

$$\frac{\partial loss}{\partial w_{kj}} = \sum_{i \in r(i,j)=1} (\sum_{l=1}^{d} x_{il} w_{lj} - y_{ij}) x_{ik}$$

则梯度为：

$$\Delta x = r.(xw - y)w^T$$

$$\Delta w = x^T[(xw - y).r]$$

其中：

```
.表示点乘法，无则表示矩阵相乘
上标T表示矩阵转置
```

梯度 `python` 代码如下：

```
x_grads = np.dot(np.multiply(record,np.dot(x,w)-y),w.T)
w_grads = np.dot(x.T,np.multiply(record,np.dot(x,w)-y))
```

然后再进行梯度下降:

```
#梯度下降
for i in range(n_iter):
    # 计算x和w的梯度
    x_grads = np.dot(np.multiply(record,np.dot(x,w)-y),w.T)
    w_grads = np.dot(x.T,np.multiply(record,np.dot(x,w)-y))
    # 更新参数
    x = alpha*x - lr*x_grads
    w = alpha*w - lr*w_grads
```

其中：

```
n_iter:训练轮数
lr:学习率
alpha：权重衰减系数，用来防止过拟合
```

10.4 动手实现基于矩阵分解的协同过滤

```
# -*- coding: utf-8 -*-

import numpy as np

def recommend(userID,lr,alpha,d,n_iter,data):
    '''
    userID(int):推荐用户ID
    lr(float):学习率
    alpha(float):权重衰减系数
    d(int):矩阵分解因子
    n_iter(int):训练轮数
    data(ndarray):电影评分表
    '''
    #获取用户数与电影数
    m,n = data.shape
    #初始化参数
    x = np.random.uniform(0,1,(m,d))
    w = np.random.uniform(0,1,(d,n))
    #创建评分记录表，无评分记为0，有评分记为1
    record = np.array(data>0,dtype=int)
    #梯度下降，更新参数
    for i in range(n_iter):
        x_grads = np.dot(np.multiply(record,np.dot(x,w)-data),w.T)
        w_grads = np.dot(x.T,np.multiply(record,np.dot(x,w)-data))
        x = alpha*x - lr*x_grads
        w = alpha*w - lr*w_grads
    #预测
    predict = np.dot(x,w)
    #将用户未看过的电影分值从低到高进行排列
    for i in range(n):
        if record[userID-1][i] == 1 :
            predict[userID-1][i] = 0
    recommend = np.argsort(predict[userID-1])

    #分数最高的5部电影
    a = recommend[-1]
    b = recommend[-2]
    c = recommend[-3]
    d = recommend[-4]
    e = recommend[-5]

    print('为用户%d推荐的电影为：\n1:%s\n2:%s\n3:%s\n4:%s\n5:%s。'\
          %(userID,movies_df['title'][a],movies_df['title'][b],movies_df['title'][c],movies_df['title'][d],mo
vies_df['title'][e]))
```

10.5 实战案例

电影评分数据

本次使用电影评分数据为 `672` 个用户对 `9123` 部电影的评分记录，部分数据如下：

userId	movieRow	rating
1	30	2.5
7	30	3
31	30	4
32	30	4

其中：

```
userId：用户编号
movieRow:电影编号
rating:评分值
```

如:

- 第一行数据表示用户 `1` 对电影 `30` 评分为 `2.5` 分。
- 第二行数据表示用户 `7` 对电影 `30` 评分为 `3` 分。

然后，我们还有电影编号与电影名字对应的数据如下：

movieRow	title
0	Toy Story (1995)
1	Jumanji (1995)
2	Grumpier Old Men (1995)
3	Waiting to Exhale (1995)

其中：

```
movieRow：电影编号
title：电影名称
```

数据下载连接 提取码：ve3v

构造用户-电影评分矩阵

大家已经知道，要使用基于矩阵分解的协同过滤算法，首先得有用户与电影评分的矩阵，而我们实际中的数据并不是以这样的形式保存，所以在使用算法前要先构造出用户-电影评分矩阵， `python` 实现代码如下：

```
import numpy as np
```

```
import pandas as pd

# 读取电影数据
ratings_df = pd.read_csv('data.csv')

# 获取用户数
userNo = max(ratings_df['userId'])+1
# 获取电影数
movieNo = max(ratings_df['movieRow'])+1

# 创建电影评分表
rating = np.zeros((userNo,movieNo))
for index,row in ratings_df.iterrows():
    rating[int(row['userId']),int(row['movieRow'])]=row['rating']
```

构造出表格后，我们就能使用上一关实现的方法来对用户进行电影推荐了：

```
# 使用上一节中实现的推荐算法进行推荐
recommend(1,1e-4,0.999,20,100,rating)
>>>
为用户1推荐的电影为：
1:Rumble Fish (1983)
2:Aquamarine (2006)
3:Stay Alive (2006)
4:Betrayal, The (Nerakhoon) (2008)
5:Midnight Express (1978)。

# 使用上一节中实现的推荐算法进行推荐
recommend(666,1e-4,0.999,20,100,rating)
>>>
为用户666推荐的电影为：
1:Aquamarine (2006)
2:It's a Boy Girl Thing (2006)
3:Kill the Messenger (2014)
4:Onion Field, The (1979)
5:Wind Rises, The (Kaze tachinu) (2013)。

# 使用上一节中实现的推荐算法进行推荐
recommend(555,1e-4,0.999,20,100,rating)
>>>
为用户555推荐的电影为：
1:Return from Witch Mountain (1978)
2:Hitcher, The (2007)
3:Betrayal, The (Nerakhoon) (2008)
4:Listen to Me Marlon (2015)
5:World of Tomorrow (2015)。

# 使用上一节中实现的推荐算法进行推荐
recommend(88,1e-4,0.999,20,100,rating)
>>>
为用户88推荐的电影为：
1:Now, Voyager (1942)
2:Betrayal, The (Nerakhoon) (2008)
3:Aquamarine (2006)
4:Post Grad (2009)
5:Hitcher, The (2007)
```