目录

第一章 绪论

1.1 大数据与数据挖掘

大数据没那么神秘

最近几年不管大小企业、国企、私企、民企到处都在说大数据，各种的以大数据为名头的会议、活动也比比皆是，你方唱罢我登台，非常热闹。例如在网上搜“大数据会议”这个关键词，能看到今年已经举办了 1000 多场次关于大数据的会议和活动，可见热度非比寻常。

虽然很多公司宣称自己是大数据公司，但实际上呢？什么样的数据叫大，是 G 级还是 T 级、 P 级、 E级、 Z级，还是 B 级？这个真的不太好定义，既然无法定义，那么反复强调自己的大数据，未免滑稽。其实，在国内，除了一些一二线互联网企业，许多宣称自己是大数据公司的数据量连 T 级都不到。在业界也有这样一种调侃的说法，一块硬盘可以打包公司所有数据的公司叫硬盘公司，一个皮包可以装载公司一切的叫皮包公司。

那么所谓的大数据是什么呢？其实就是数据仓库与数据挖掘。而且早在美国的 90 年代就已经有了这两个概念，现在只不过把它们两合在一起变得更加新潮了而已。

很多人都在宣传，大数据给相关公司带来业绩上翻天覆地的变化，而实际情况呢？如果不能很好地对数据做数据挖掘的话，大数据不但不能能给相关公司带来业绩上的任何变化，反而还会因为大量冗余数据给公司运维带来麻烦。数据只是死的，如果你不能从中找出有价值的内容，再“大”也没意义。其实数据量无论大小，如果能够很好地从数据中挖掘出一些有用的知识，那么就是非常有意义的。

所以说，大数据并没有那么神秘。

什么是数据挖掘

接下来，来介绍一下什么是数据挖掘。什么是数据挖掘？为了回答这个问题，有很多数据挖掘书籍都会提到一个经典案例："尿布与啤酒"的故事。

在一家超市里，有一个有趣的现象：尿布和啤酒赫然摆在一起出售。但是这个奇怪的举措却使尿布和啤酒的销量双双增加了。这不是一个笑话，而是发生在美国沃尔玛连锁店超市的真实案例，并一直为商家所津津乐道。沃尔玛拥有世界上最大的数据仓库系统，为了能够准确了解顾客在其门店的购买习惯，沃尔玛对其顾客的购物行为进行购物篮分析，想知道顾客经常一起购买的商品有哪些。沃尔玛数据仓库里集中了其各门店的详细原始交易数据。在这些原始交易数据的基础上，沃尔玛利用数据挖掘方法对这些数据进行分析和挖掘。一个意外的发现是："跟尿布一起购买最多的商品竟是啤酒！经过大量实际调查和分析，揭示了一个隐藏在"尿布与啤酒"背后的美国人的一种行为模式：在美国，一些年轻的父亲下班后经常要到超市去买婴儿尿布，而他们中有 30%～40% 的人同时也为自己买一些啤酒。产生这一现象的原因是：美国的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他们喜欢的啤酒。按常规思维，尿布与啤酒风马牛不相及，若不是借助数据挖掘技术对大量交易数据进行挖掘分析，沃尔玛是不可能发现数据内在这一有价值的规律的。

相信你也看出来了，这就是数据挖掘，从常人的知识外找到线索，或者从五花八门的数据中找出一些潜在规律。通俗说，数据挖掘可以做到以下几点：

- 找到没有意识到的问题
- 找到未来发展的趋势
- 找到过去存在的问题
- 把定性的问题定量化
- 数据对象关联的规则问题
- 找到一些隐藏的资料

可想而知，这几点的威力是十分巨大的。而且在我们的日常生活中无时无刻都在享受着数据挖掘为我们带来的便利。

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中特征值的分布。所谓的分布，就是查看数据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。

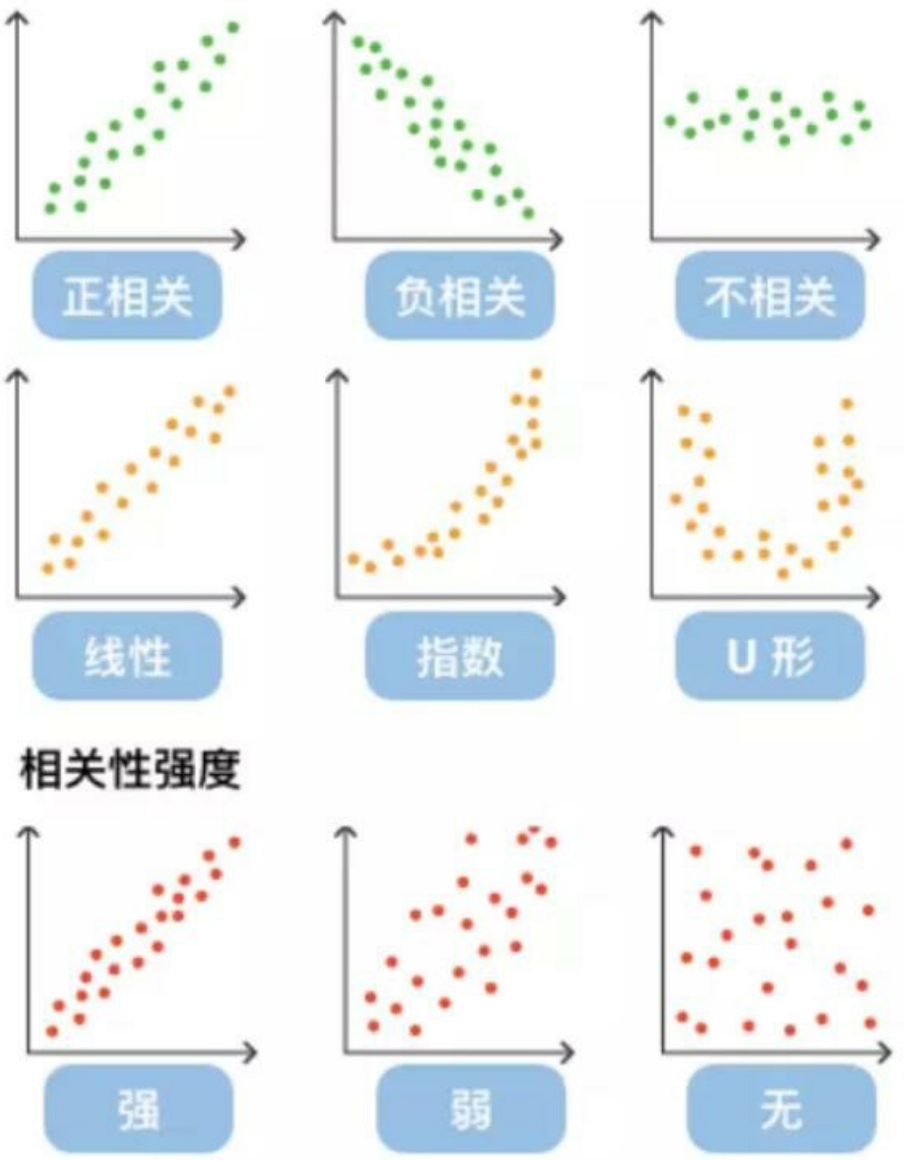
2.3 数据可视化

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。

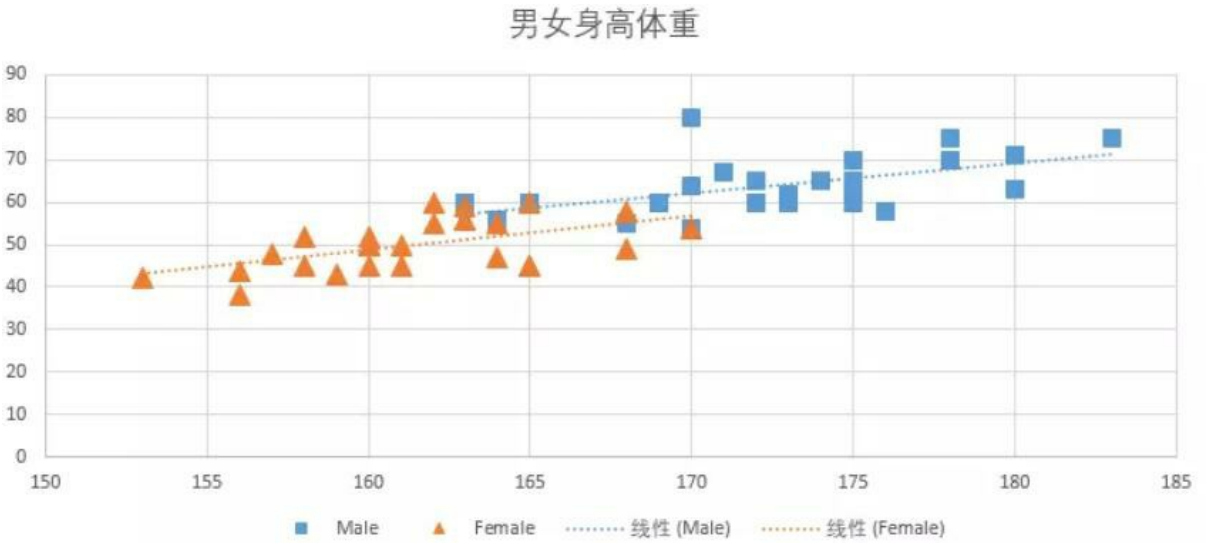
通过观察散点图上数据点的分布情况，我们可以推断出变量间的相关性。如果变量之间不存在相互关系，那么在散点图上就会表现为随机分布的离散的点，如果存在某种相关性，那么大部分的数据点就会相对密集并以某种趋势呈现。数据的相关关系主要分为：正相关（两个变量值同时增长）、负相关（一个变量值增加另一个变量值下降）、不相关、线性相关、指数相关等，表现在散点图上的大致分布如下图所示。那些离点集群较远的点我们称为离群点或者异常点。



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：

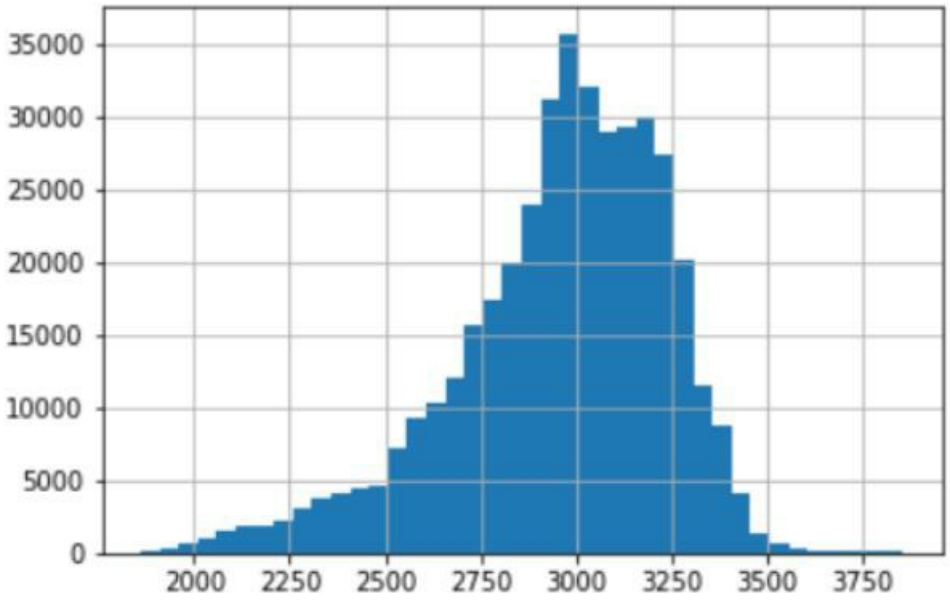


从可视化结果可以看出，不管是男生还是女生仿佛都遵循这一个规律，就是身高越高体重就越重。而且细心一点，可以看出身高 170 体重为 90 的人应该是属于肥胖了。

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。

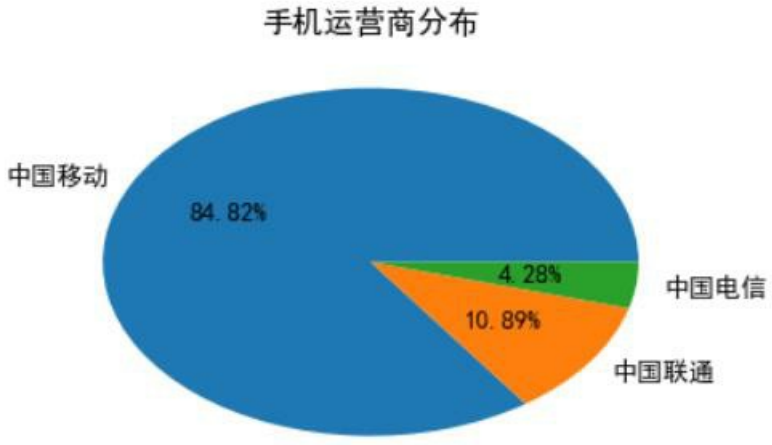
例如这是罗斯福国家公园树木数据集中树木所处地的海拔高度的直方图。



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 数据预处理的重要性

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。为什么这么说呢？不妨看一下下面这样一个故事情节。

比如有一天你的 boss 找到你说：XX 听说你对数据挖掘很熟悉啊，正好我们公司有很多 xx 方面的数据，你看看能不能做一个数据挖掘的项目为我们公司提供一些决策参考。你听到这里是高兴还是悲伤，具体因人而异（要是小弟听到了，绝对会很高兴）。我这里假设你很高兴，接到 boss 的圣旨以后你就屁颠屁颠的找公司的相关数据。毕竟手持 boss 的圣旨，所以数据获取应该不是太难，拿到数据你就开始疯狂的想使用什么模型呢？kNN，决策树，线性回归等等模型，你通过“认真”思考后选择了一个模型，迫不及待的把数据往里面喂。当你信心满满的点击 run 后，你会看到下面一行，一行，一行的红色字体，大体意思是这里数字无效，那里数据为空等等，这时候你的内心可能是崩溃的。

虽然仅仅是我编出来的一个故事场景，但是当我们真正地做数据挖掘时，在整个工程中，数据预处理所花费的精力是最多的。就好比，你是一个堪比中华小当家一样的名厨，当你拿到上等的新鲜食材时，你肯定能不费吹灰之力地做出美味佳肴。若尼拿到的是发酸发臭，品相不好的食材，你可能费了九牛二虎之力才能勉强赶上食堂大叔的做菜水准。

因此，数据预处理的效果有多好，基本上就决定了你数据挖掘的效果有多好。

在本章中，将介绍几种数据预处理的常用技巧的意义，以及如何使用 `sklearn` 来实现这些常用技巧。

3.2 数据预处理常用技巧---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。这是因为有些数据挖掘算法中需要计算距离或梯度等信息，而这些信息需要数据中的特征处于同一量纲内才会有比较好的效果。

比如现在问你个问题，是 1000 克的棉花重，还是 1 公斤的铁重？你肯定能不假思索的回答：一样重！但是，有些数据挖掘算法会觉得 1000 克的棉花重，因为它并不会看 1000 后面的单位是克还是公斤。所以如果想要使用这类算法来进行数据挖掘，那么首先要做的事情就是对数据进行标准化，因为标准化能够解决特征量纲不一致的问题。

Z-score标准化

这种方法基于原始数据的均值和标准差进行数据的标准化。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：

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

`sklearn` 的 `preprocessing` 模块中的函数 `scale` 实现了 `Z-score` 标准化的功能。实例代码如下：

```
# 导入preprocessing模块
from sklearn import preprocessing
# 导入numpy库
import numpy as np

# 定义数据，该数据中有3条记录，每条记录有3个特征
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

# 对数据进行z-score标准化，并将结果保存至X_scaled
X_scaled = preprocessing.scale(X_train)

# z-score标准化的结果
>>>X_scaled
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

从结果上看，可以看出 `scale` 函数好像起了作用，为了验证标准化是否正确，我们可以检查一下 `X_scaled` 的均值和方差。代码如下：

```
# 计算X_scaled中每个特征的均值，发现全是0
>>> X_scaled.mean(axis=0)
array([ 0.,  0.,  0.])
```

```
# 计算X_scaled中每个特征的方差，发现全是1
>>> X_scaled.std(axis=0)
array([ 1.,  1.,  1.])
```

嗯，不错，`scale` 成功地对数据进行了标准化。

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：

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

`sklearn` 的 `preprocessing` 模块中的 `MinMaxScaler` 类的 `fit_transform` 函数实现了 `Min-max` 标准化的功能。实例代码如下：

```
# 导入preprocessing模块
from sklearn import preprocessing
# 导入numpy库
import numpy as np

# 定义数据，该数据中有3条记录，每条记录有3个特征
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

# 实例化MinMaxScaler对象
min_max_scaler = preprocessing.MinMaxScaler()
# 对数据进行Min-max标准化，并将结果保存至X_train_minmax
X_train_minmax = min_max_scaler.fit_transform(X_train)

# Min-max标准化的结果
>>> X_train_minmax
array([[ 0.5       ,  0.        ,  1.        ],
       [ 1.        ,  0.5       ,  0.33333333],
       [ 0.        ,  1.        ,  0.        ]])
```

可以看出，经过 `Min-max` 标准化后，数据中的所有特征值都缩放到了 `[0, 1]` 的区间内。

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：

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

`sklearn` 的 `preprocessing` 模块中的 `MaxAbsScaler` 类的 `fit_transform` 函数实现了 `MaxAbs` 标准化的功能。实例代码如下：

```
# 导入preprocessing模块
from sklearn import preprocessing
# 导入numpy库
import numpy as np

# 定义数据，该数据中有3条记录，每条记录有3个特征
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

# 实例化MaxAbsScaler对象
max_abs_scaler = preprocessing.MaxAbsScaler()
# 对数据进行MaxAbs标准化，并将结果保存至X_train_maxabs
X_train_maxabs = max_abs_scaler.fit_transform(X_train)

# MaxAbs标准化的结果
>>> X_train_maxabs
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
```

可以看出，经过 `MaxAbs` 标准化后，数据中的所有特征值都缩放到了 `[-1, 1]` 的区间内。

3.3 数据预处理常用技巧---归一化

归一化与标准化

归一化是缩放单个样本以具有单位范数的过程。也就是说，上一节中提到的标准化是对数据中的每一列进行计算，而本节中提到的归一化则是对数据中的每一行进行计算。

一般来说，当我们想要计算两个样本之间的相似度之前，可以尝试一下使用归一化来作为计算相似度之前的预处理，因为有可能会使得相似度计算地更加准确。

在实践中，用的最多的是**L1**范式归一化和**L2**范式归一化，下面来详细看看这两种归一化是怎样计算的。

L1范式归一化

`L1` 范式定义如下：

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

表示数据 `x` 中每个特征值 `xi` 的绝对值之和。

因此， `L1` 范式归一化就是将样本中每个特征的特征值除以 `L1` 范式。虽然这个功能实现起来简单，但我们也不必重新造轮子。在 `sklearn` 中已经提供了该功能，即 `normalize` 函数，用法如下：

```
# 导入normalize函数
from sklearn.preprocessing import normalize

# 定义数据，数据中有3条样本，每条样本中有3个特征
data = np.array([[-1,0,1],
                 [1,0,1],
                 [1,2,3]])

# 使用normalize函数进行L1范式归一化
# data即想要归一化的数据， l1即表示想要使用L1范式归一化来进行归一化处理
data = normalize(data,'l1')

# 可以分析一下：第一行的L1范数是不是2，在归一化完了之后，第一行第一列的值是不是-0.5
>>>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：

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

$$\sqrt{\sum_{i=1}}$$

表示数据 `x` 中每个特征值 `xi` 计算平方和再开跟。

因此， `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` 中已经提供了该功能，即 `normalize` 函数，用法如下：

```
# 导入normalize函数
from sklearn.preprocessing import normalize

# 定义数据，数据中有3条样本，每条样本中有3个特征
data = np.array([[-1,0,1],
                 [1,0,1],
                 [1,2,3]])

# 使用normalize函数进行L2范式归一化
# data即想要归一化的数据， l2即表示想要使用L2范式归一化来进行归一化处理
data = normalize(data,'l2')

# 可以分析一下：第一行的L1范数是不是根号2，在归一化完了之后，第一行第一列的值是不是-0.707
>>>data
array([[-0.707,  0.   ,  0.707],
       [ 0.707,  0.   ,  0.707],
       [ 0.267,  0.535,  0.802]])
```

3.4 数据预处理常用技巧---离散值编码

离散值编码的目的

想要知道离散值编码的目的，那么首先需要知道什么是离散值。离散值的概念其实你并不陌生，因为早在第 2 章第 1 节中你就已经接触到了离散值的概念，没错！就是离散特征！

要知道，不管是数据挖掘还是机器学习算法，它们都只认识数字，一到碰到诸如字符串之类的数据时就会向你抱怨：我太难了！

所以当我们碰到带有离散值的数据时，我们就要想办法来把这些离散值进行转换，转换成算法所认识的数字。

在进行离散值编码时，通常会尝试使用如下两种方法，`LabelEncoder` 和 `OneHotEncoder`。下面，我们会试图使用这两种编码方法来解决数据中的离散值编码问题。

LabelEncoder

这种方法会对特征中所出现的离散值建立一种映射关系。这种映射关系很简单，例如数据中有 `Sex` 这一特征，并且该特征中的值的值要么是 `male`，要么是 `female`。如果我们对 `Sex` 特征使用 `LabelEncoder` 方法进行离散值编码的话，可能会得到一个映射关系: `{'male':0, 'female':1}`，即当看到 `'male'` 时就将其改成 `0`，看到 `'female'` 时将其改成 `1`。

因此进行编码后的结果为：

Sex
0
1
1
1
0

这样，就成功地将字符串类型的特征值转换成了数值类型的特征值。

当然， `sklearn` 已经为我们提供了实现该功能的接口: `LabelEncoder` ，使用案例如下：

```
# 导入LabelEncoder类
from sklearn.preprocessing import LabelEncoder

# 定义数据
label = ['male','female']

# 实例化LabelEncoder对象
int_label = LabelEncoder()

# 进行离散值编码
label = int_label.fit_transform(label)

# 成功将male转换成0，将female转换成1
>>>label
array([0, 1])
```

OneHotEncoder

虽然 `LabelEncoder` 能将代表不同意义的字符串转换成数字，但是在转换之后很可能会带入顺序信息。比如将 `‘male’` 映射成 `0` ， `'female'` 映射成 `1` 时，有可能会使得数据挖掘算法认为 `'female'` 更重要，或者 `‘male’` 和 `'female'` 之间存在着一定的大小关系。

我们知道，现在是男女平等的时代，不应该出现男权或者女权的现象。那应该怎样避免将特征值分成三六九等呢？我们可以使用 `OneHotEncoder` 方法来进行离散值编码。

`OneHotEncoder` 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：

sex
male
male
female
male
female

那么经过 `OneHotEncoder` 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`):

sex_male	sex_female
1	0
1	0
0	1
1	0
0	1

你会发现，经过 `OneHotEncoder` 编码后， `sex` 特征变成了无序的二进制特征。而且能够轻松的看出第一条样本的性别是 `male` ，第三条样本的性别是 `female` 。

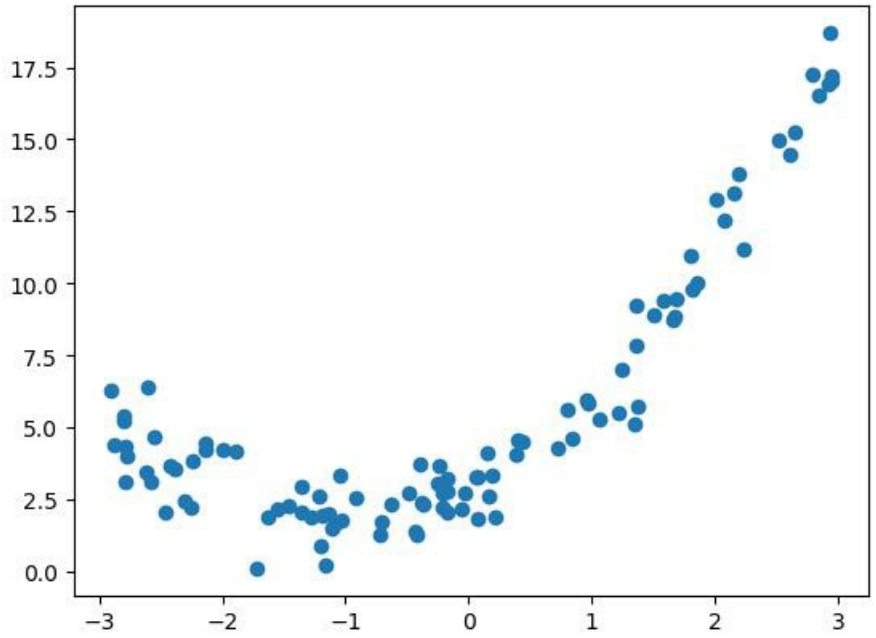
当然， `sklearn` 已经为我们提供了实现该功能的接口: `OneHotEncoder` ，使用案例如下：

```
import numpy as np
# 导入OneHotEncoder类
from sklearn.preprocessing import OneHotEncoder


# 定义数据
label = ['male','female']


# 实例化OneHotEncoder对象
onehot_label = OneHotEncoder()

# 对数据进行编码
label = onehot_label.fit_transform(label)

# 样本的特征变成了2个，分别表示male和female
>>>label
array([[1, 0],
       [0, 1]])
```

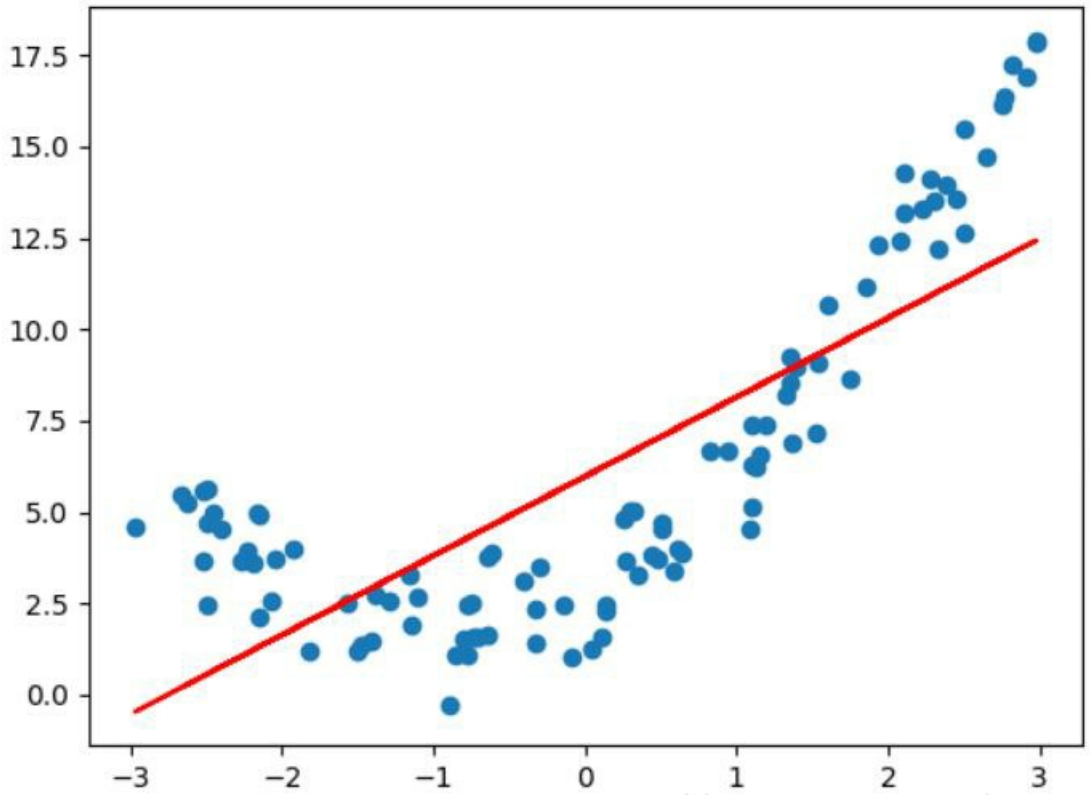
3.5 数据预处理常用技巧---生成多项式特征

引例

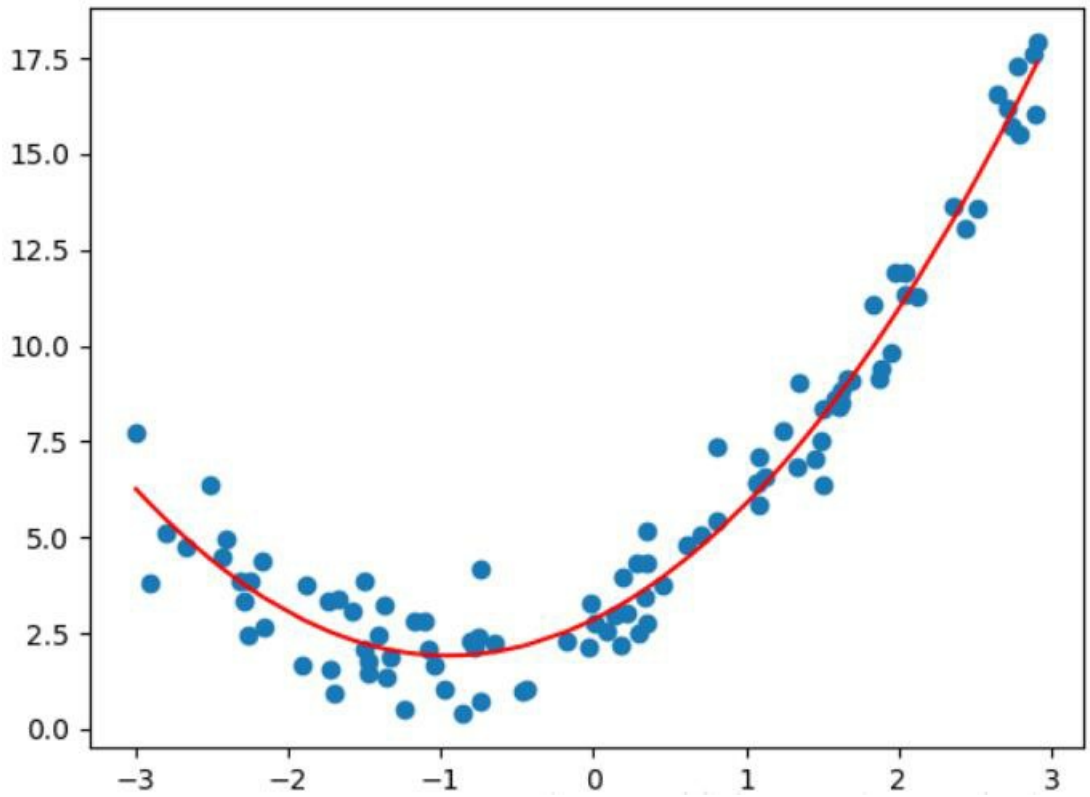
假设现在给了你一份数据，数据的分布如下图所示(其中横轴代表神秘特征 `x` 的值，纵轴代表与 `x` 所对应的目标值 `y`，也就是说纵轴的值可以根据 `x` 以及一种映射关系计算出来)：



我想让你根据 `x` 来将 `y` 算出来，此时，你观察数据的分布，认为数据的分布有点像一条直线，所以你可能会使用一条直线来来表示这些数据的分布，如下图中红色的直线：



但是你会发现，红色的直线好像并不能很好的表示数据的分布，毕竟数据的分布像是一个弯弯的勾号，如果能根据特征 `x`，使得这条直线能“打勾”就好了。如下图所示：



这个时候，我们就可以试试生成多项式特征了。即将原来的“直线”给“掰弯”。

使用**sklearn**来生成多项式特征

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

```
import numpy as np
# 导入PolynomialFeatures类
from sklearn.preprocessing import PolynomialFeatures

# 定义数据:
# [[0, 1]
# [2, 3]
# [4, 5]]
data = np.arange(6).reshape(3, 2)

# 实例化PolynomialFeatures的对象，多项式的阶为2阶
poly = PolynomialFeatures(degree=2)

# 添加多项式特征
data = poly.fit_transform(data)

# 添加之后的结果
>>>data
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

你会看到原始数据中只有 2 个特征，而添加完多项式特征之后，特征数量增加到了 6 个。这是因为在生成多项式特征时会将原始的两个特征(x_1, x_2)，先扩展成$(1, x_1, x_2)$，记为 X ，然后将两个 X 中的元素两两相乘，相乘后会得到$1, x_1, x_2, x_1x_2, x_1^2, x_2^2$，即转换过程如下图所示：

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

所以特征数量会增加到 6 个。但是在一些情况下，我们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
# 导入PolynomialFeatures类
from sklearn.preprocessing import PolynomialFeatures

# 定义数据:
# [[0, 1]
# [2, 3]
# [4, 5]]
data = np.arange(6).reshape(3, 2)

# 实例化PolynomialFeatures的对象，多项式的阶为2阶，并只保留交叉项
poly = PolynomialFeatures(degree=2, interaction_only=True)

# 添加多项式特征
data = poly.fit_transform(data)

# 添加之后的结果
>>>data
```

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

特征转换情况如下：

所以 `interaction_only` 为 `True` 时，特征的数量只增加到了 4 。

3.6 数据预处理常用技巧---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数据挖掘算法兼容，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。例如我有这样一份数据：

性别	工资
男	
女	
女	5000
男	
男	6000
女	

如果我们将带有缺失值的记录全部删掉的话，那么数据中就只有一条样本可以使用了，这时你会发现，就 2 条样本我挖掘什么啊？

因此更好的处理缺失值的策略是，想办法将缺失值给补上，虽然有点亡羊补牢的意思，但是总比破罐子破摔强。

使用sklearn来处理缺失值

在 `sklearn` 中提供了接口 `Imputer` 来帮助我们快速的对缺失值进行填充。使用方法如下：

```
# 导入Imputer类
from sklearn.preprocessing import Imputer

# 定义数据，np.nan表示缺失
data = [[np.nan, 2], [6, np.nan], [7, 4],[np.nan,4]]

# 实例化Imputer对象，missing_values表示我们将什么值看成是缺失，strategy表示按照什么样的方式来填充缺失值

# strategy='mean'表示填充缺失值时，使用列的均值来填充
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)

# 填充缺失值
data = imp.fit_transform(data)

# 根据数据可知，第一列的均值是6.5，第二列的均值是3.33333，所以数据中缺失的部分就是使用的均值来进行填充的
>>>data
array([[6.5       , 2.        ],
       [6.        , 3.33333333],
```

```
       [7.        , 4.        ],
       [6.5       , 4.        ]])
```

当然，`Imputer`的填充策略不止`mean`这一种，还有两种策略如下：

- `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。

第四章 使用k近邻算法检测红酒品质

4.1 问题的本质

红酒鉴定家的烦恼

身为一为红酒鉴定师，每天打交道打得最多的肯定就是红酒啦。一为合格的红酒鉴定师能根据红酒的颜色、气味、口感、酿造时间、木桶的材质等一系列信息准确的推测出该红酒的品质。

但是人总是喜欢偷懒，当需要在短时间内鉴定大量红酒的品质时，就算是久经沙场的红酒鉴定师，也会觉得心有余而力不足。

如果我们能够使用一种方法，该方法能够根据红酒品质的历史数据来推测出现在需要鉴定的红酒的品质的话，那么红酒鉴定师就可以悠闲地坐在沙发上喝咖啡了。

下面是红酒鉴定师收藏多年的红酒数据中的冰山一角，数据中包含了红酒的颜色，酸碱度，酒精度等信息。其中 `quality` 表示红酒的品质(该特征是离散特征)，数字的值越大，表示红酒的品质越高。

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

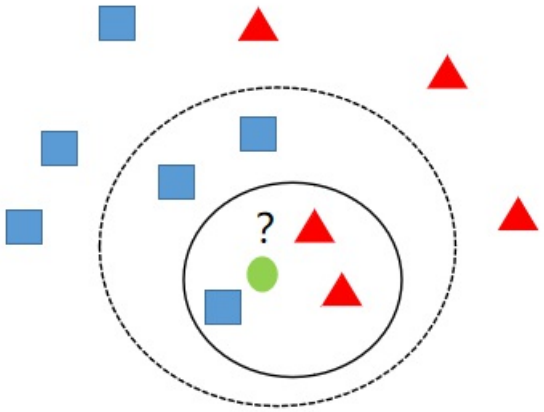
抓住主要矛盾

现在我们想要得到一种算法，自动的能根据红酒的颜色、酸碱度、酒精度等信息来推算出该红酒的品质。而品质这一属性是一种离散值。假设红酒的品质分为 1 到 9 ，9 个等级。那么我们的算法要的事情就是输出 1 到 9 之间的一个数字即可。像这种需要输出离散值的算法，我们通常称之为分类算法。(如果将 9 个等级看成是 9 种类别，可以思考一下算法的输出是不是相当于在做分类的事情。)

作为正式接触数据挖掘算法的第一章，本章的算法应该尽量简单，有效。所以本章将使用k近邻算法来完成红酒品质检测的功能。因为k近邻算法的思想是众多数据挖掘算法中最简单的，非常适合入门。如果你准备好了，请继续往下看。

4.2 k近邻算法原理

古人云：“近朱者赤，近墨者黑”。意思是说如果一个人身边的朋友中坏人比较多，那么这个人很可能也是坏人，如果身边的朋友中好人比较多，那么他很可能也是个好人。其实k近邻算法的核心思想就是这一句古话。



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。

也就是说，如果三角形是好人，正方形是坏人，那么绿色的圆就是好人，因为与绿色的圆关系最紧密的三个人中有两个是好人。

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。

同样，如果三角形是好人，正方形是坏人，那么绿色的圆就是坏人，因为与绿色的圆关系最紧密的五个人中有三个是坏人。

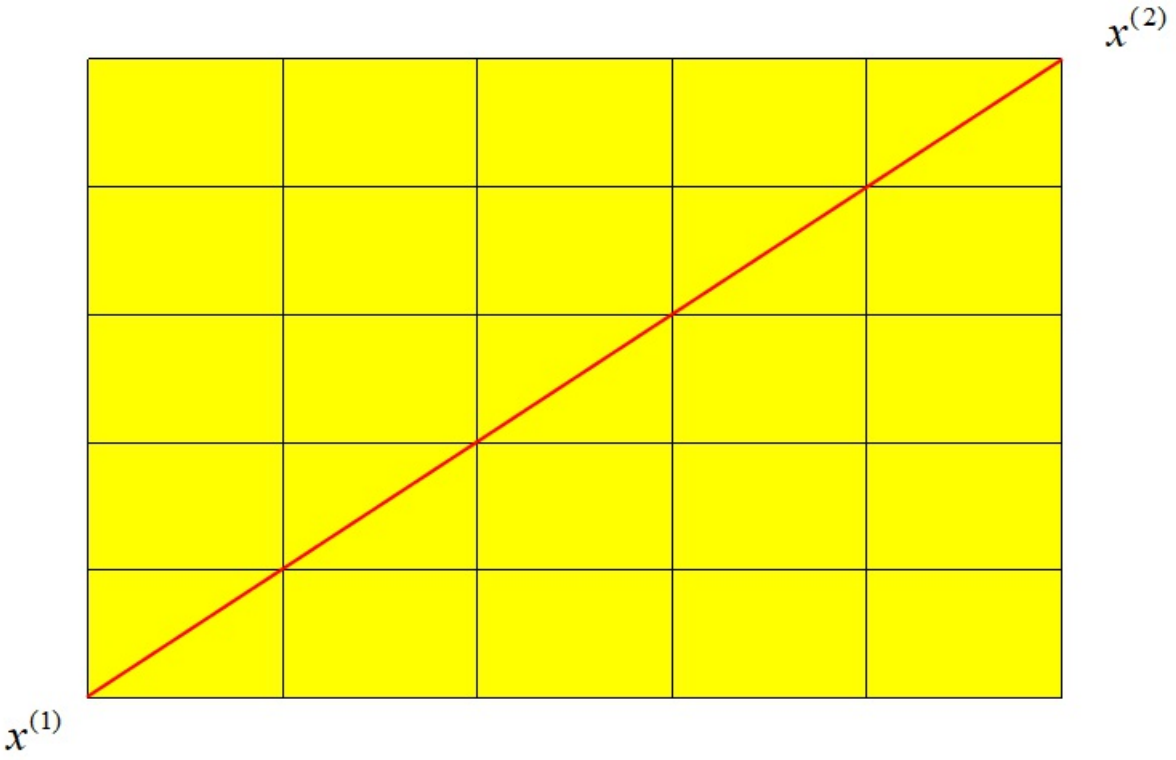
OK，现在我们已经知道，如何判别一个样本属于哪个类型，主要是看离它最近的几个样本中哪个类型的数量最多，则该样本属于数量最多的类型。这里，存在两个问题：

- 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。



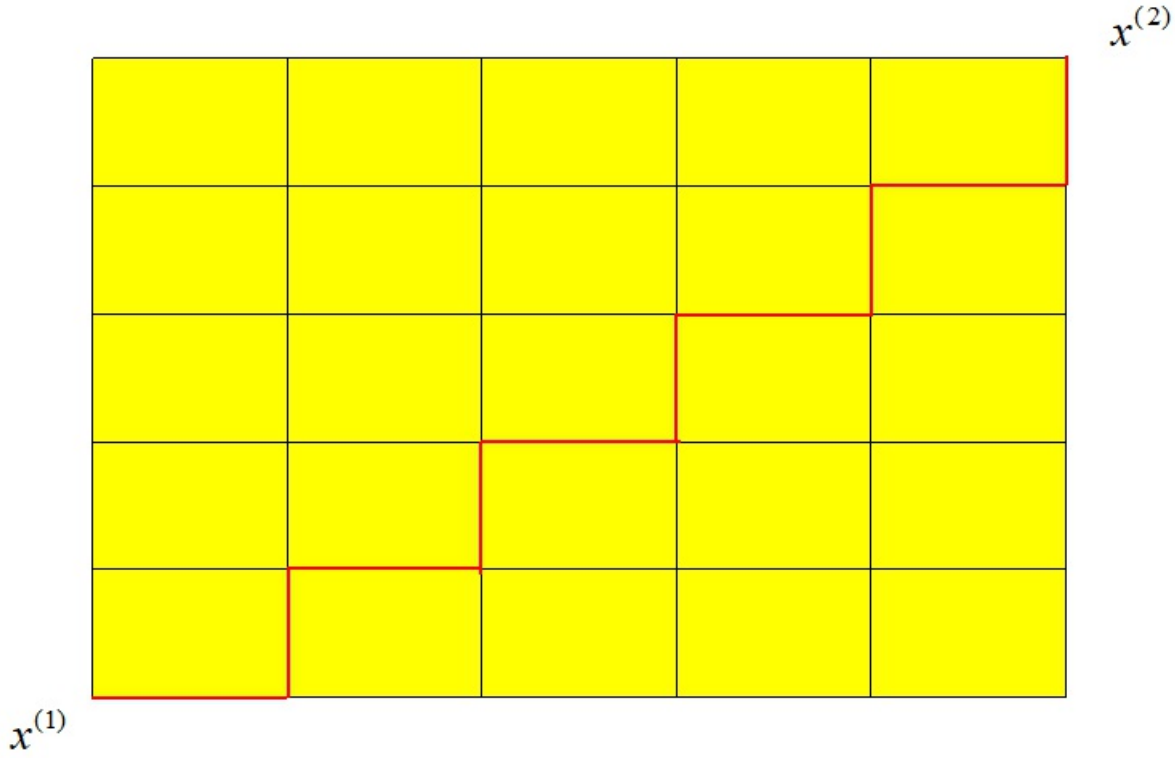
二维平面上欧式距离计算公式：

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

`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：

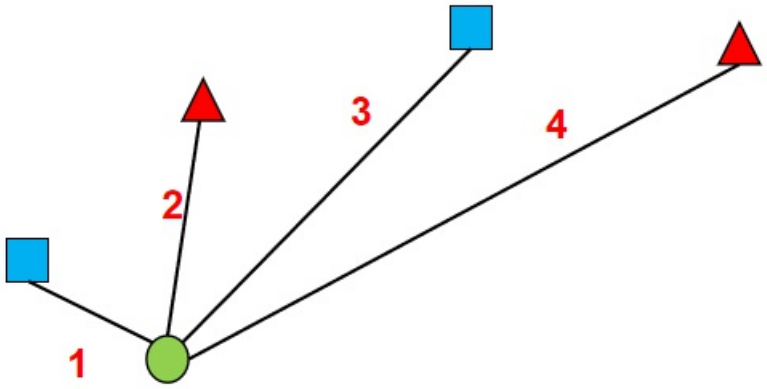
$$d_{12} = |x_1^{(1)} - x_1^{(2)}| + |x_2^{(1)} - x_2^{(2)}|$$

`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

k近邻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 动手实现k近邻

k近邻算法非常简单，可以很方便的使用 `python` 实现，实现代码如下：

```
#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 检测红酒品质

数据预处理

在使用k近邻算法进行检测之前，我们可以先查看一下各个特征的均值和标准差。

```
[1.30006180e+01 2.33634831e+00 2.36651685e+00 1.94949438e+01 9.97415730e+01 2.29511236e+00 2.02926966e+00 3.6
1853933e-01 1.59089888e+00 5.05808988e+00 9.57449438e-01 2.61168539e+00 7.46893258e+02]

[8.09542915e-01 1.11400363e+00 2.73572294e-01 3.33016976e+00 1.42423077e+01 6.24090564e-01 9.96048950e-01 1.2
4103260e-01 5.70748849e-01 2.31176466e+00 2.27928607e-01 7.07993265e-01 3.14021657e+02]
```

从打印结果可以看出，有的特征的均值和标准差都比较大，例如如最后一个特征。如果现在用k近邻算法来对这样的数据进行分类的话，k近邻算法会认为最后一个特征比较重要。因为假设有两个样本的最后一个特征值分别为 `1` 和 `100`，那么这两个样本之间的距离可能就被这最后一个特征决定了。这样就很有可能会影响k近邻算法的准确度。为了解决这种问题，我们可以对数据进行标准化。标准化的知识，以及如何实现标准化在第三章已经向你介绍过了，相信你对这些知识已经很熟悉了，在这里就不多赘述了。

检测品质

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中的红数据进行品质检测了：

```
predict = knn_clf(3,train_feature,train_label,test_feature)
predict
>>>array([6, 5, 4, 7, 4, 3, 6, 4, 5, 4, 9, 2, 7, 8, 4, 6, 9, 5, 7, 4, 7, 5,
       8, 6, 0, 9, 6, 4, 5, 7, 5, 9, 8, 6, 4, 8, 8, 5, 5, 6, 7, 9, 4, 6,
       8, 7, 6, 7, 4, 5, 4, 5, 4, 2, 4, 7, 0, 5, 5, 5, 4, 6, 7, 8, 5, 6,
       8, 5, 5, 4, 6, 3, 7, 4, 3, 4, 6, 9, 7, 7, 8, 5, 6, 5, 6, 5, 5, 4,
       9, 7, 7, 6, 8, 8, 6, 8, 5, 7, 4, 6, 8, 8, 5, 4, 6, 5, 6, 9, 9, 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k近邻算法检测红酒品质，正确率能达到 `99%` 以上。此时此刻，我相信，红酒鉴定师的烦恼要从活太多干不完，转变成以后可能没活干了。

第五章 使用线性回归算法预测房价

5.1 什么是回归

在上一章介绍了如何使用k近邻算法来对红酒的品质进行检测。由于红酒的品质是一种离散值，所以，红酒品质检测这件事说白了，就是将对红酒的品质进行分类。

那么细心的你可能会发现一个问题，如果我们想要的输出是连续值而非离散值，该怎么办呢？

那这个时候，就可以引出一个概念了，那就是回归。那回归是什么意思呢？其实说白了，就是这个我们的算法的输出的结果是个连续的值。如果输出不是个连续值而是个离散值那就叫分类。那什么叫做连续值呢？非常简单。

举个栗子：比如我告诉你我这里有间房子，这间房子有 40 平，在地铁口，然后你来猜一猜我的房子总共值多少钱？这就是连续值，因为房子可能值80万，也可能值 80.2 万，也可能值 80.111 万。再比如，我告诉你我有间房子，120 平，在地铁口，总共值 180 万，然后你来猜猜我这间房子会有几个卧室？那这就是离散值了。因为卧室的个数只可能是 1， 2， 3，4，充其量到 5 个封顶了，而且卧室个数也不可能是什么 1.1 ， 2.9 个。所以呢，对于数据挖掘新手来说，你只要知道我要完成的任务是预测一个连续值的话，那这个任务就是回归。是离散值的话就是分类。

所以预测房价其实就是一种回归问题。而接下来要介绍的线性回归算法就是回归算法中的一种。

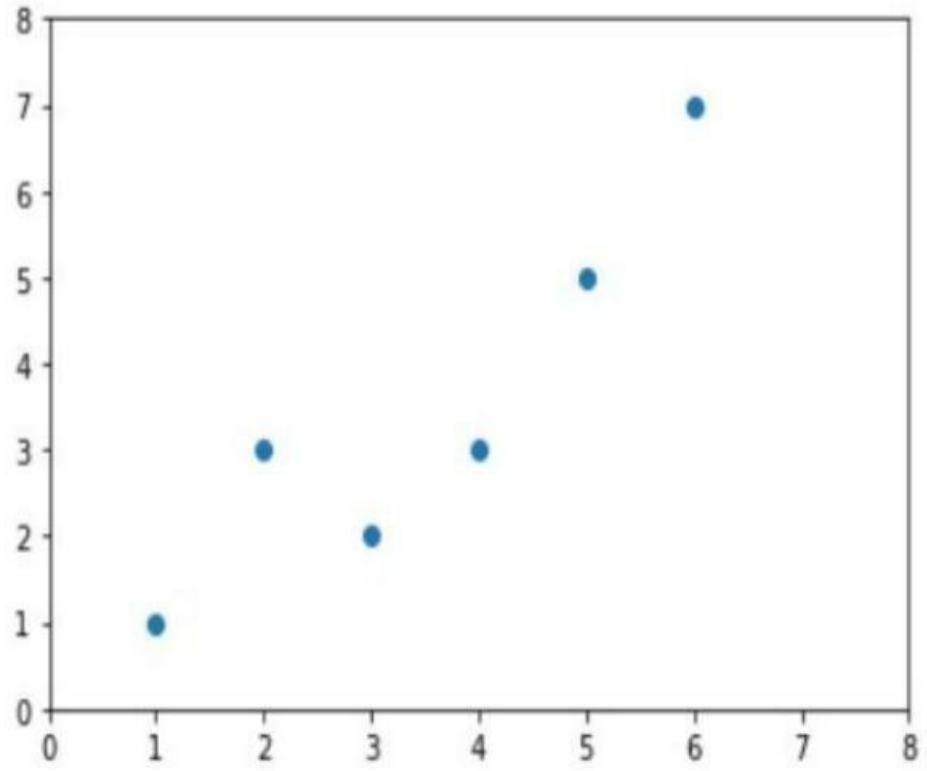
5.2 线性回归算法原理

什么是线性回归

线性回归是什么意思？可以拆字释义。回归肯定不用我多说了，那什么是线性呢？我们可以回忆一下初中时学过的直线方程：`y=k*x+b`

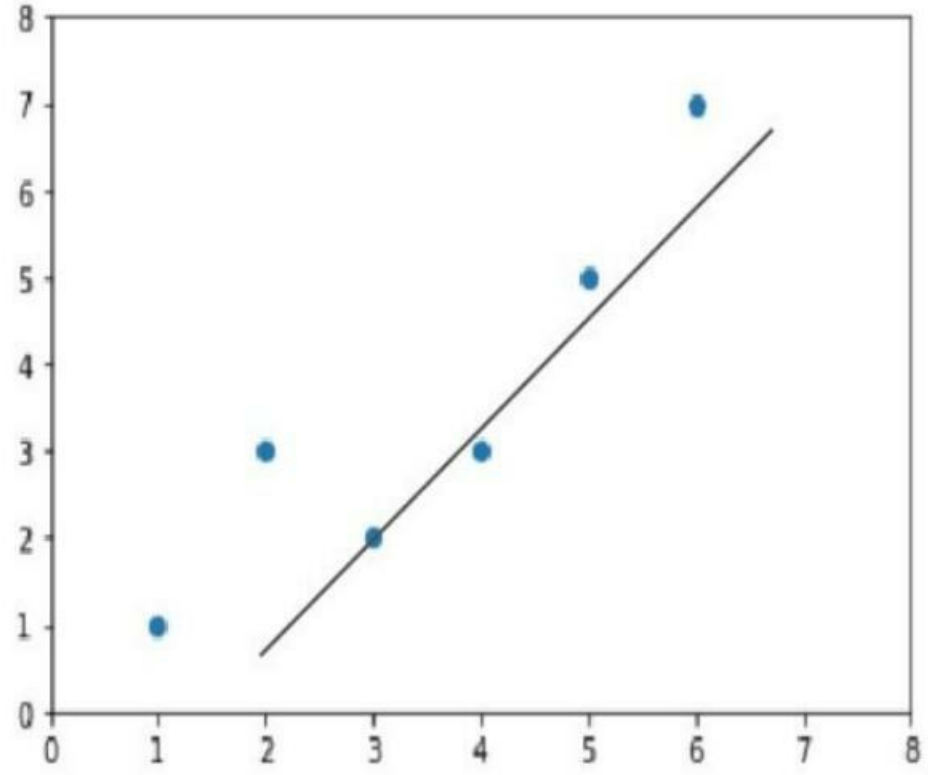
这个式子表达的是，当我知道 k （参数）和 b （参数）的情况下，我随便给一个 x 我都能通过这个方程 算出 y 来。而且呢，这个式子是线性的，为什么呢？因为从直觉上来说，你都知道，这个式子的函数图像 是条直线。

从理论上来说，这式子满足线性系统的性质。你可能会觉得疑惑，这一节要说的是线性回归，我说个这么 low 直线方程干啥？其实，说白了，线性回归就是在 N 维空间中找一个形式像直线方程一样的函数来拟合数据而已。比如说，我 现在有这么一张图，横坐标代表房子的面积，纵坐标代表房价。

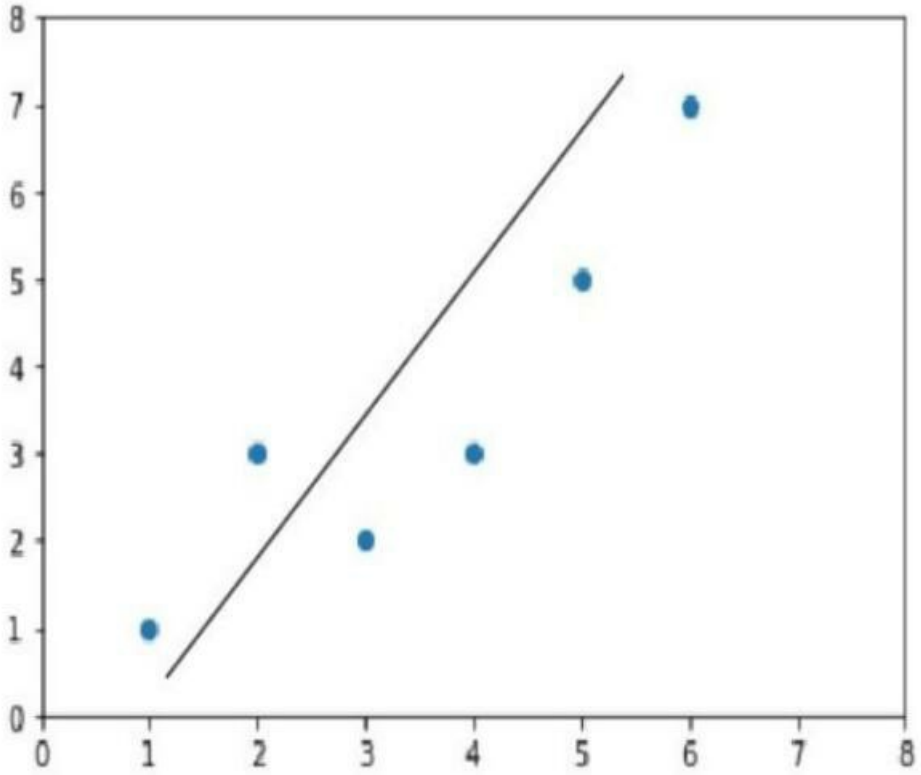


然后呢，线性回归就是要找一条直线，并且让这条直线尽可能地拟合图中的数据点。

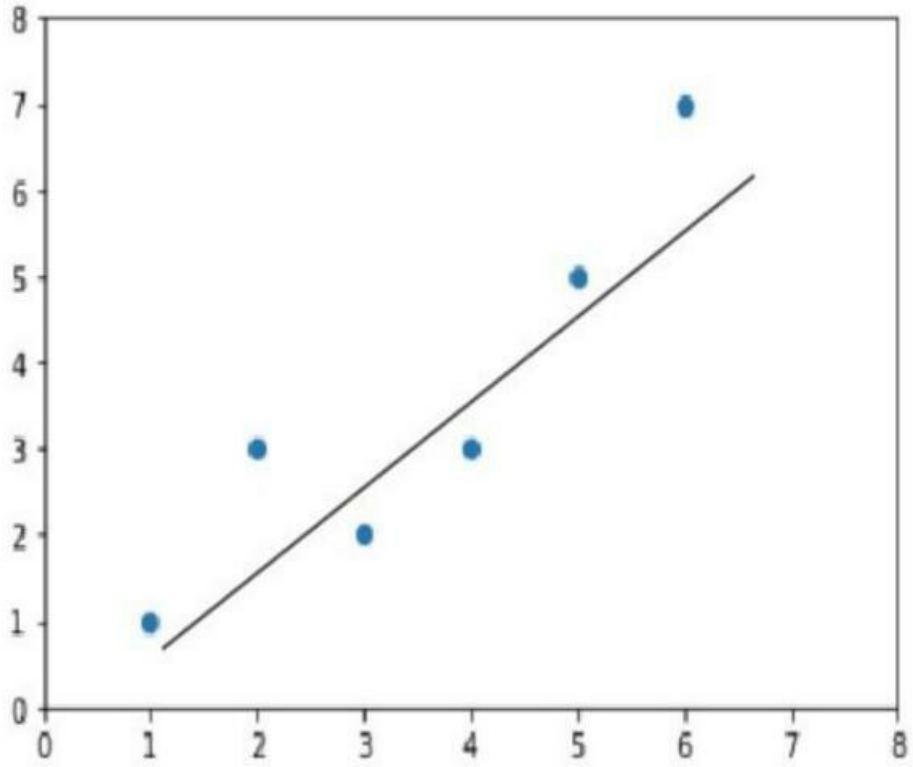
那如果让 1000 位朋友来找这条直线就可能找出 1000 种直线来，比如这样



这样



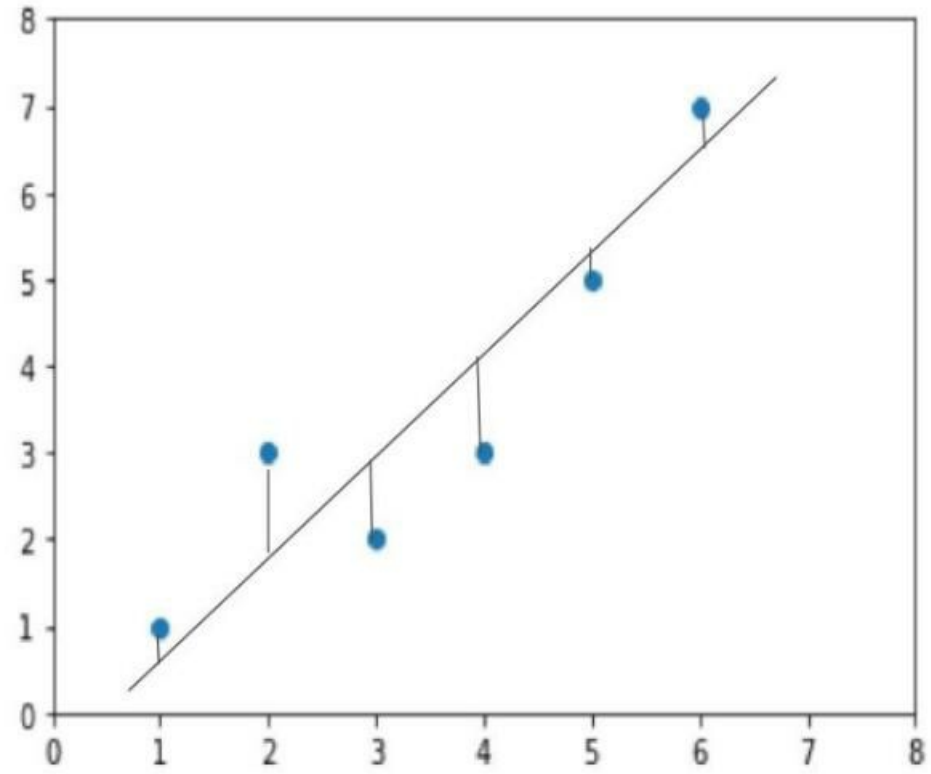
或者这样



喏，其实找直线的过程就是在做线性回归，只不过这个叫法更有高大上而已。

损失函数

那既然是找直线，那肯定是要有一个评判的标准，来评判哪条直线才是最好的。道理我们都懂，那咋评判呢？其实只要算一下实际房价和我找出的直线根据房子大小预测出来的房价之间的差距就行了。说白了就是算两点的距离。当把所有实际房价和预测出来的房价的差距（距离）算出来然后做个加和，就能量化出现在预测的房价和实际房价之间的误差。例如下图中我画了很多条小数线，每一条小数线就是实际房价和预测房价的差距（距离）。



然后把每条小竖线的长度加起来就等于现在通过这条直线预测出的房价与实际房价之间的差距。那每条小竖线的长度的加和怎么算？其实就是欧式距离加和，公式为：$\sum_{i=1}^{m}(y^{(i)}-\hat{y}^{(i)})^2$(其中$y(i)$表示的是实际房价，$\hat{y}(i)$表示的是预测房价)。

这个欧氏距离加和其实就是用来量化预测结果和真实结果的误差的一个函数。在机器学习中称它为损失函数（说白了就是计算误差的函数）。那有了这个函数，就相当于有了一个评判标准，当这个函数的值越小，就越说明找到的这条直线越能拟合房价数据。所以说啊，线性回归就是通过这个损失函数做为评判标准来找出一条直线。

如果假设$h_{(\theta)}(x)$表示当权重为θ，输入为x时计算出来的$\hat{y}(i)$，那么线性回归的损失函数$J(\theta)$就是：

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^i)-y^i)^2$$

怎样计算出线性回归的解？

现在你应该已经弄明白了一个事实，那就是我只要找到一组参数（也就是线性方程每一项上的系数）能让我的损失函数的值最小，那我这一组参数就能最好的拟合我现在的训练数据。

那怎么来找到这一组参数呢？其实我们可以使用线性回归的正规方程解，这名字听起来高大上，其实本质就是根据一个固定的式子计算出参数。线性回归的正规方程解是这样算出来的。

假设数据中 `m` 条记录，每条记录中有 `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

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

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

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

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 动手实现线性回归

知道线性回归的正规方程解的公式之后，我们可以很容易的使用 `python` 实现线性回归算法。 `python` 实现代码如下：

```
#encoding=utf8
import numpy as np

# 线性回归算法
def lr(train_feature,train_label,test_feature):
    '''
    input:
        train_feature(ndarray):历史数据
        train_label(ndarray):每条历史数据所对应的答案
        test_feature(ndarray):测试数据
    output:
        predict(ndarray):每条测试数据所对应的预测结果
    '''
    #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
```

5.4 预测房价

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，因为我们需要人为的划分出两个部分，一部分代表历史数据，另一部分代表未来的数据。用于检验我们线性回归算法的准确度。划分数据，我们可以使用 `sklearn` 中提供的 `train_test_split` 函数来实现，示例代码如下：

```
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表示历史数据
# test_feature表示未来数据
# train_label表示每条历史记录所对应的答案
# test_label表示每条未来记录所对应的答案
train_feature,test_feature,train_label,test_label = train_test_split(x,y,test_size=0.2,random_state=666)
```

预测房价

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：

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

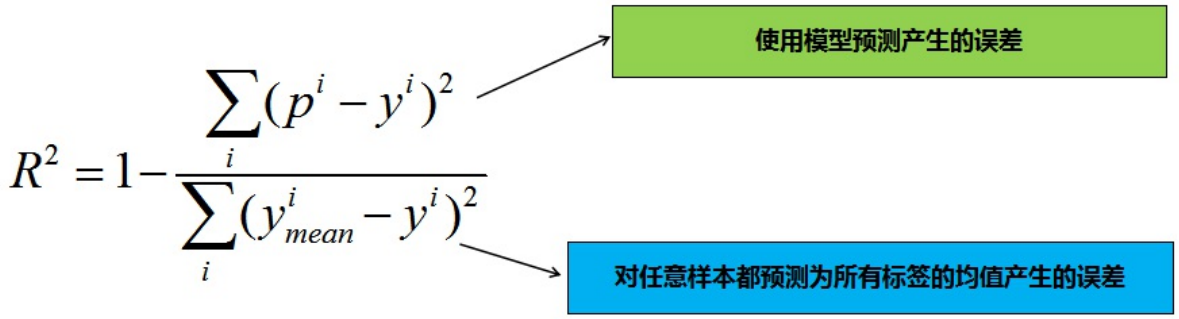
`MAE` 虽然不作为损失函数，确是一个非常直观的评估指标，它表示每个样本的预测标签值与真实标签值的 `L1` 距离。

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
```

可以看到，我们已经能够根据房价的历史数据，来预测未知房屋的价格了，而且预测的准确度也是可以接受的。

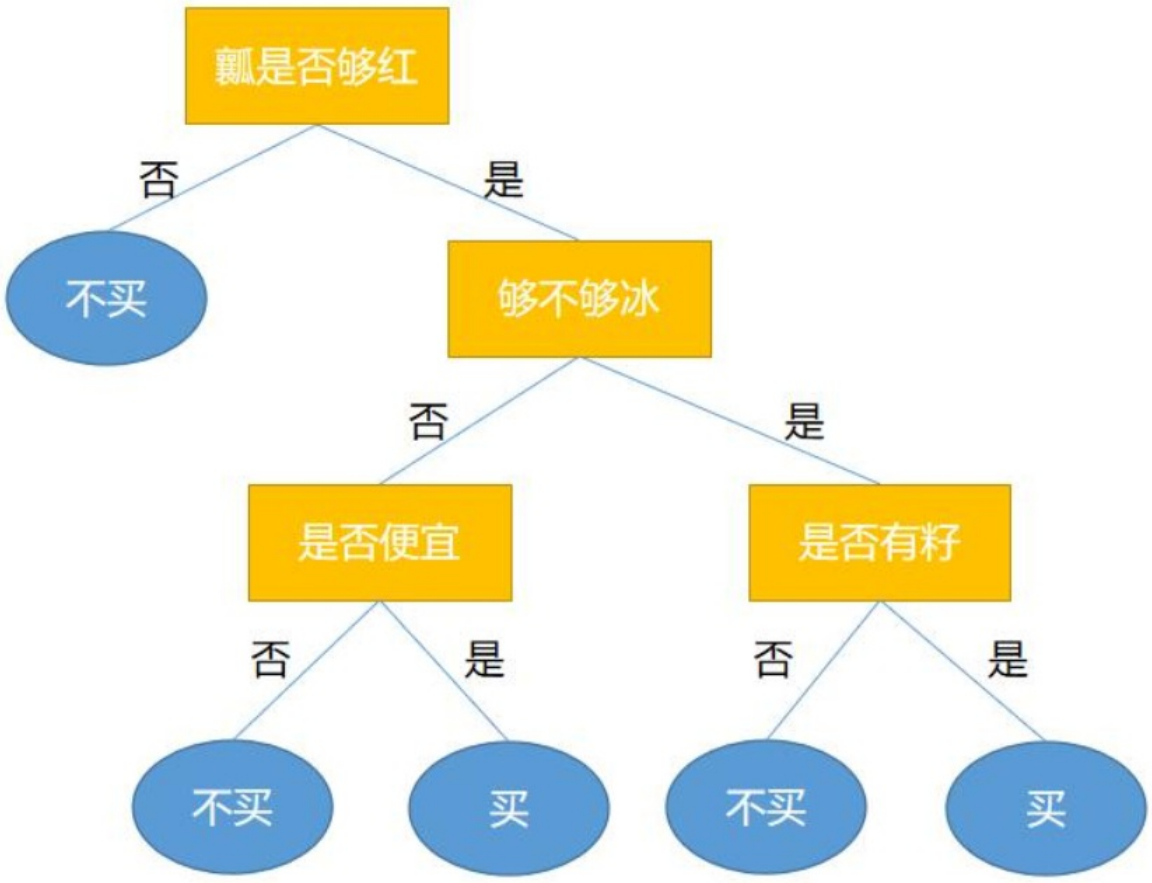
第六章 使用决策树算法识别花朵

6.1 决策树算法的核心思想

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

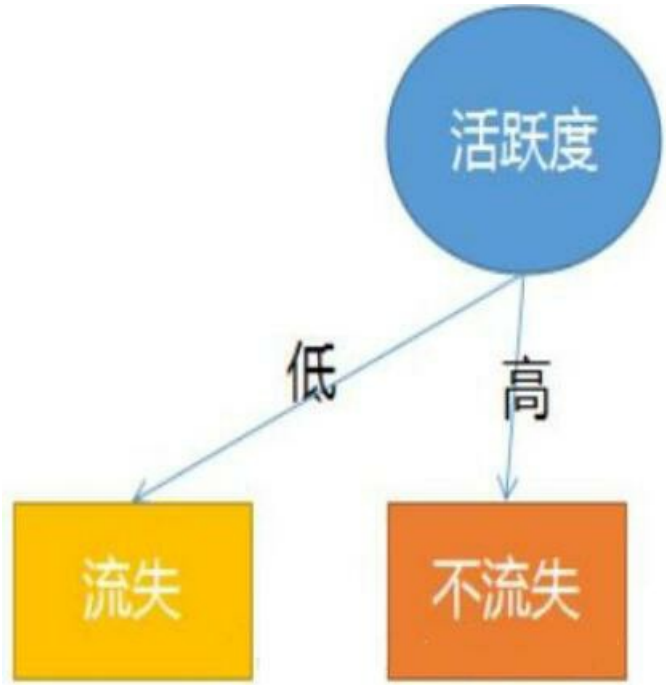
那信息增益算出来之后有什么意义呢？回到读心术的问题，为了我能更加准确的猜出你心中所想，我肯定是问的问题越好就能猜得越准！换句话来说我肯定是要想出一个信息增益最大（减少不确定性程度最高）的问题来问你。其实 `ID3` 算法也是这么想的。 `ID3` 算法的思想是从训练集 `D` 中计算每个特征的信息增益，然后看哪个最大就选哪个作为当前结点。然后继续重复刚刚的步骤来构建决策树。

6.3 决策树构建流程

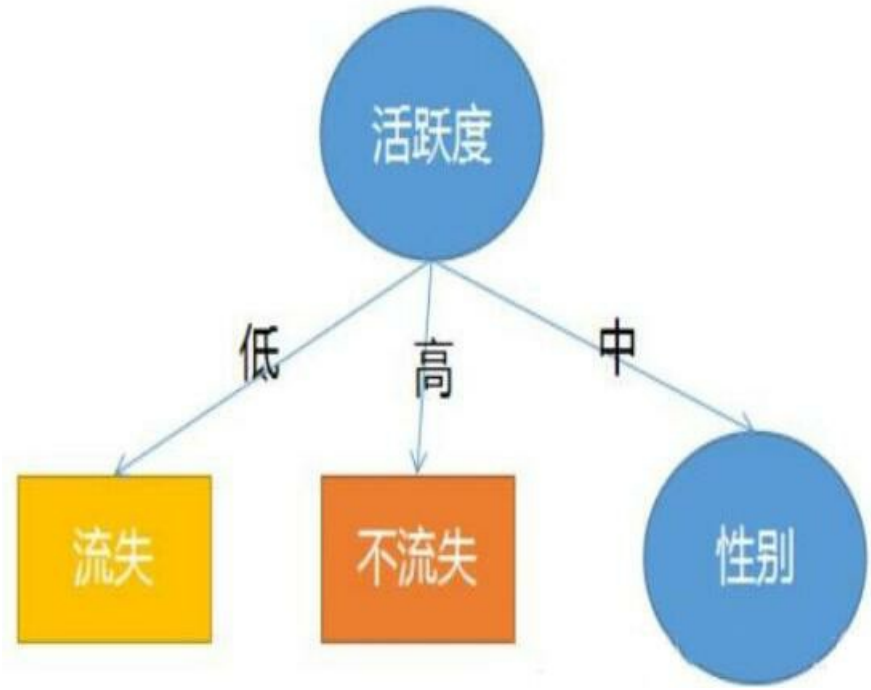
接下来将通过使用上一节中客户流失的数据来描述一下 `ID3` 算法构建决策树的过程。

一开始我们已经算过信息增益最大的是活跃度，所以决策树的根节点是活跃度 。所以这个时候树是这样的：

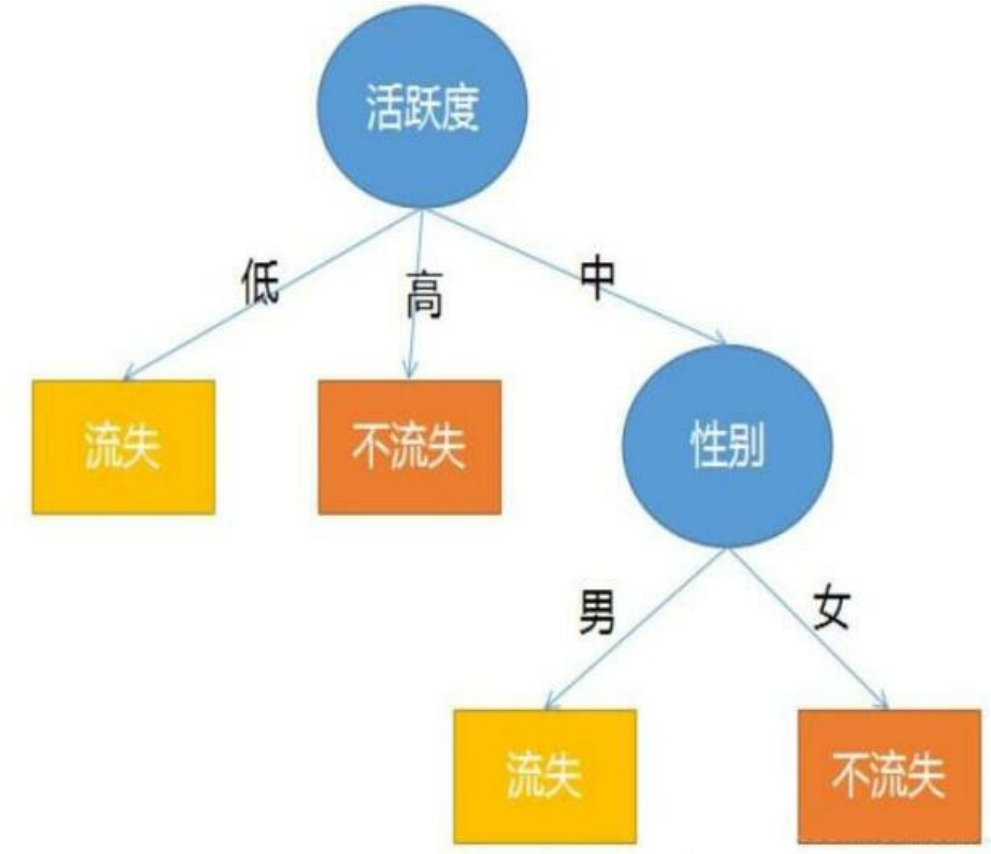
然后发现训练集中的数据表示当我活跃度低的时候一定会流失，活跃度高的时候一定不流失，所以可以先在根节点上接上两个叶子节点。



但是活跃度为中的时候就不一定流失了，所以这个时候就可以把活跃度为低和为高的数据屏蔽掉，屏蔽掉之后 5 条数据，接着把这 5 条数据当成训练集来继续算哪个特征的信息增益最高，很明显算完之后是性别这个特征，所以这时候树变成了这样：



这时候呢，数据集里没有其他特征可以选择了（总共就两个特征，活跃度已经是根节点了），所以就看我性别是男或女的时候那种情况最有可能出现了。此时性别为男的用户中有 1 个是流失，1 个是不流失，五五开。所以可以考虑随机选个结果当输出了。性别为女的用户中有全部都流失，所以性别为女时输出是流失。所以呢，树就成了这样：



好了，决策树构造好了。从图可以看出决策树有一个非常好的地方就是模型的解释性非常强！！很明显，如果现在来了一条数据 (男, 高) 的话，输出会是不流失。

6.4 动手实现决策树

理解了 `ID3` 算法之后，我们可以尝试使用 `python` 来实现该算法了。由于 `ID3` 算法在构建决策树的时候需要计算信息增益。所以首先可以使用如下代码，实现信息增益的计算。实现思路是分别实现熵和条件熵的计算，再根据信息增益的公式实现信息增益的计算。

```
#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
    '''
    # 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

        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
```

有了计算信息增益的功能后，我们还需要一个功能，就是需要找到数据中信息增益最高的特征是哪个，这就需要一个函数来计算出信息增益最高的特征的索引。实现方法如下：

```
# 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
```

有了以上的这些功能以后，我们可以开始实现决策树的构造过程了。大致过程和上一节所描述的一样，实现如下：

```
#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
```

构造好决策树之后，我们就可以实现使用这棵决策树来进行预测的方法了，实现如下：

```
#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
```

OK，现在我们已经实现了决策树算法，接下来，我们来看一下怎样使用我们实现好了的算法来识别花朵。

6.5 识别花朵

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 什么是图像分割

图像是由像素组成的

这是从新浪 `2009` 年的验证码中截出来的一张图。相信大家已经看出来了是字母 `y` 。这张看似非常简单的图在计算机中却是由一个个像素所排列组成的。像素(`pixel`)是图像的组成单位，像素的值域是 `[0, 255]` 。像素值越低的话我们看起来就越黑，越高的话看起来就越白。如图所示：

0 255

正因为像素有代表明暗的属性，所以我们看到的这张图在计算机中的存储是这样的。

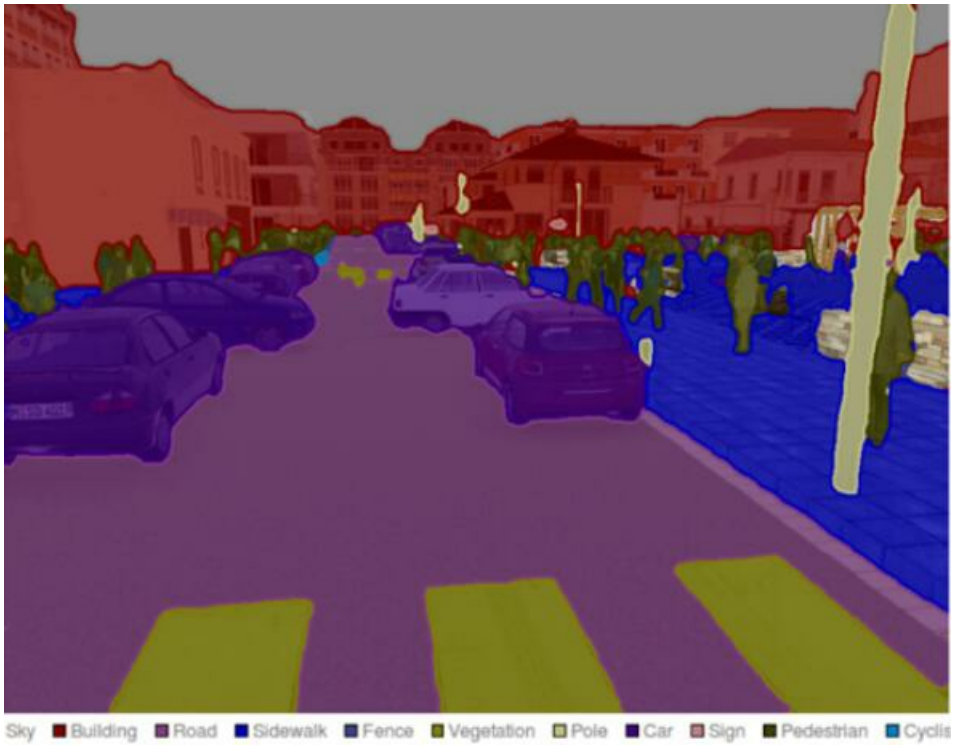
255	255	255	255	255	255	255	255	179	26	229	76	0	76	255
0	0	0	255	255	255	255	255	179	26	229	76	0	76	255
255	255	255	76	26	229	255	255	179	26	255	76	128	255	255
0	0	128	179	229	26	179	128	96	255	255	255	255	255	255
255	76	128	255	229	26	0	0	96	255	26	76	128	255	255
255	255	128	255	255	255	76	0	96	229	26	0	128	255	255
255	255	255	76	26	229	255	128	96	229	26	179	255	255	255
255	255	255	179	0	0	179	255	255	229	0	179	255	255	255
255	255	255	179	229	26	0	128	179	0	0	179	255	255	255
255	255	255	255	255	255	76	0	0	0	26	179	255	255	255
255	255	255	255	255	255	76	0	76	229	26	179	255	255	255
255	255	255	255	255	255	76	0	76	229	26	0	128	255	255
255	255	255	255	255	255	255	255	255	229	0	0	128	255	255
255	255	255	255	255	255	255	255	0	0	0	0	128	255	255
255	255	255	255	255	255	255	128	0	0	0	179	255	255	255

相信你可以站远点看能够看得出来也是一个`y`。

图像分割的意义

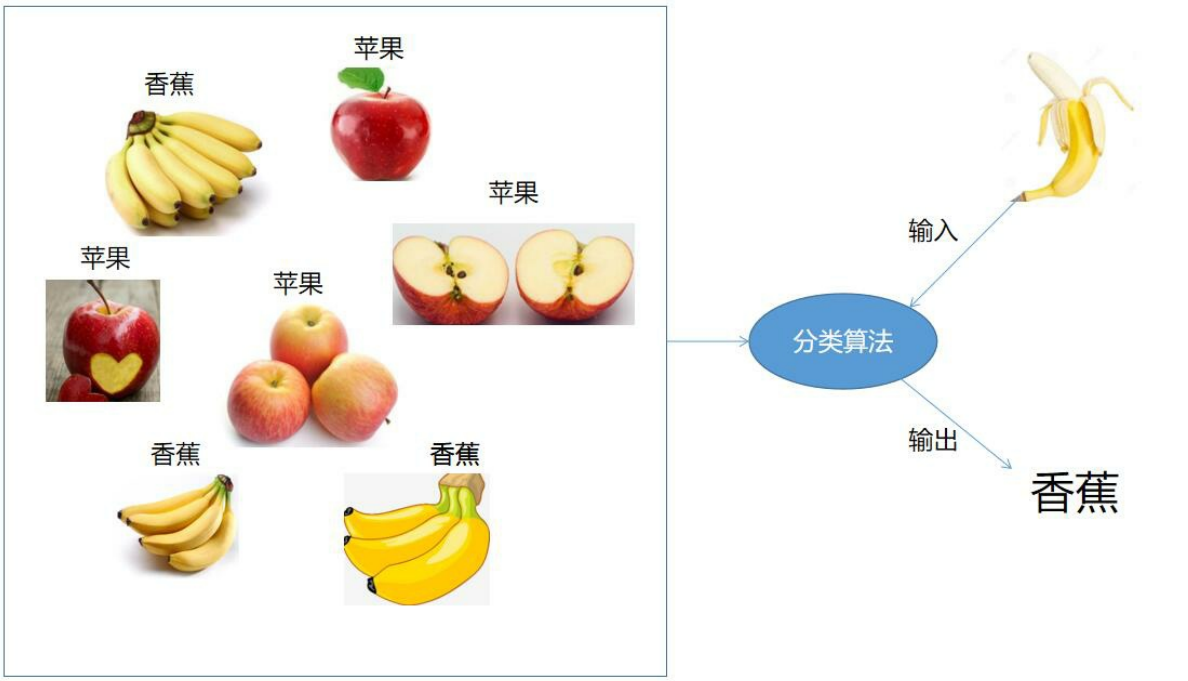
现在已经知道数字图像是由很多个像素组成的，如果图像的高和宽很大的时候，我们想识别出图像中的某个对象的时候就面临了一个问题，就是搜索范围太大了。比如下面这张图，如果我想识别出图中的建筑，那我的算法可能需要查看所有像素，很显然，这是一种非常低效的方式。

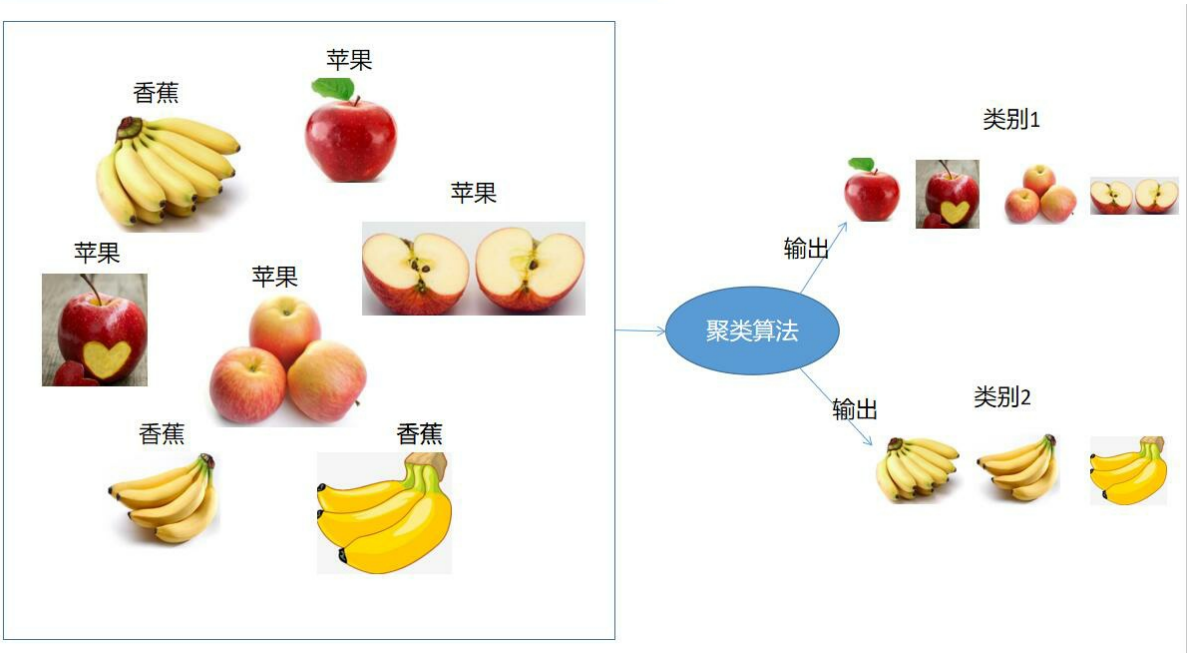
倘若现在我换个思路，我在识别建筑之前我先把图像中对象用不同的编号或者颜色来划分成不同的区域（图像处理中将其称为感兴趣区域也叫`ROI`），然后再根据我想要识别的对象来查看对象对应的`ROI`即可。这样做能很好的提高图像识别的效率，并且能够在一定程度上过滤掉一些无用的信息。比如我要识别出上图中的建筑，我可能会对整张图做图像分割，然后我只要对整张图的上半部分做图像识别就可以了。



图像分割与聚类

此时出现了一个新名词叫做聚类。其实聚类和分类是比较相近的。他们的共同点是将数据划分成好几个类别，而不同的是，分类的历史数据中是带有标准答案的；而聚类时，历史数据中是不带标准答案的。如下图所示：





也就是说，聚类算法会将觉得长得差不多的数据划分为同一种类别。这种思想与图像分割的功能是一致的，因为图像分割其实就是将觉得长得差不多的像素划分为同一种类别。所以聚类算法可是实现图像分割的功能。接下来要介绍的 k 均值算法，就是一种典型的聚类算法。

7.2 k均值算法原理

k 均值算法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 均值算法，之所以称为 k 均值 是因为它可以发现 k 个簇(即类别)，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。

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` ：

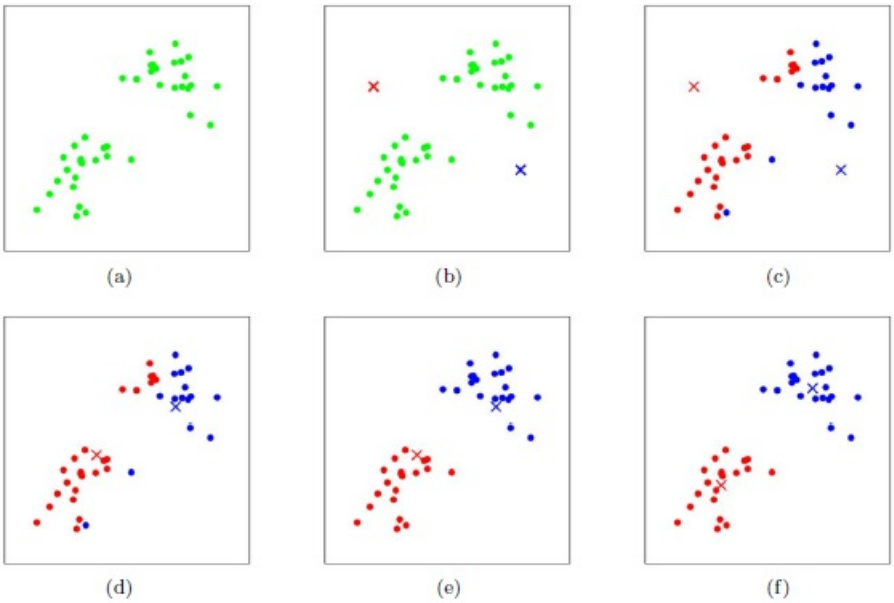
$$\sum_{i=1}^{k}\sum_{x\in c_i}(x-u_i)^2$$

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。

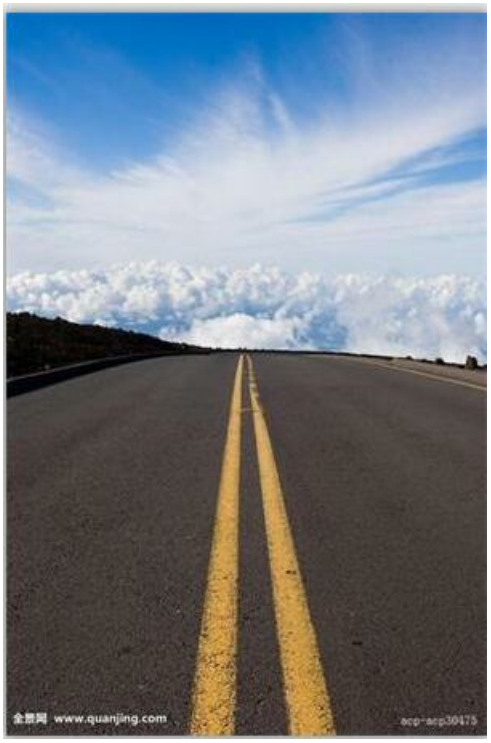
所以该算法的伪代码如下：

```
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 图像分割

了解数据

既然要对图像进行图像分割，那肯定需要有图像。在这里，为你准备了一张图像，如下图所示：



如果我们想将整张图分割成双黄线，马路，路边，天空这 4 个部分，那么很明显我们可以使用 k均值算法来进行分割，而且此时的 k 为 4。因为我们样将所有的像素聚类成 4 个簇。

代码实现

读取图像

首先我们需要将图像读入内存，`python` 有很多库都实现了读取图像的功能。在这里，我将使用 `opencv` 这个库来读取图像。`opencv` 在计算机视觉领域的使用是非常广泛的，如果你对计算机视觉感兴趣，可以深入了解一下 `opencv` 以及一些图像处理的知识，在这里我就不多做介绍了。

`opencv` 读图像很简单，只需要使用如下代码即可：

```
# 导入opencv库
import cv2

# 读取图像，图像名字为test.jpg，并将图像保存到img变量中
img = cv2.imread('test.jpg')
```

读取到图像后，就可以着手实现 k 均值算法了。

k均值算法

在实现 k 均值算法之前，可以思考一下 k 均值算法所需要的参数，很明显需要三个参数，一个是数据，在这里也就是读取到的图像；另一个就是 k，刚刚已经提到过，在这里，k 为 4。还有一个就是 k 均值算法的最大迭代次数。

因此可以写出如下函数声明：

```
def kmeans(n, k, image):
```

接下来我们来实现函数体。对于图像来说，它可以看成是一个三维数组，但是 k 均值算法需要的是一个二维数组，所以我们需要对数据进行变形。而且还需要将图像进行升维，因为在算法结束时，需要知道每个像素所对应的簇标签值。所以会有如下代码：

```
    # 图像的高
    height = image.shape[0]
    # 图像的高
    width = image.shape[1]
    tmp = image.reshape(-1, 3)
    result = tmp.copy()

    #扩展一个维度用来存放标签
    result = np.column_stack((result, np.ones(height*width)))
```

做好数据处理后，可以开始根据上一节所提到的 k 均值算法的原理来实现该算法了。首先需要初始化质心。

```
    # 初始化质心
    center_point = np.random.choice(height*width, k, replace=False)
    center = result[center_point, :]
    # 初始化距离矩阵
    distance = [[] for i in range(k)]
```

然后需要不断地迭代更新我们的质心。

```
    # 迭代n次
    for i in range(n):
        # 计算每个像素到各个质心的距离
        for j in range(k):
            distance[j] = np.sqrt(np.sum(np.square(result - np.array(center[j])), axis=1))
        # 为每个像素打上簇标签
        result[:, 3] = np.argmin(np.array(distance), axis=0)

        # 更新质心
        for j in range(k):
            center[j] = np.mean(result[result[:, 3] == j], axis=0)
    return result
```

因此，完整的代码如下：

```
def kmeans(n, k, image):
    height = image.shape[0]
```

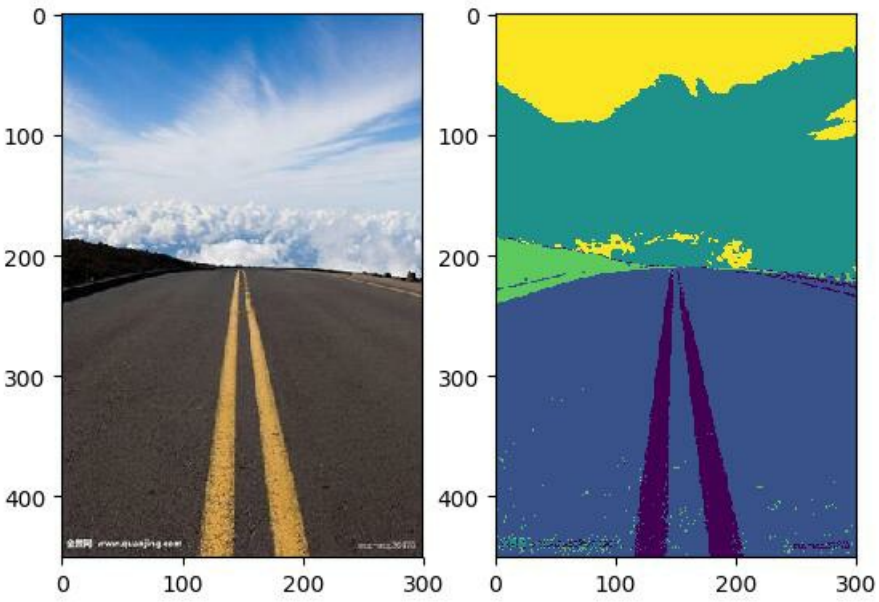
```
    width = image.shape[1]
    tmp = image.reshape(-1, 3)
    result = tmp.copy()

    #扩展一个维度用来存放标签
    result = np.column_stack((result, np.ones(height*width)))

    center_point = np.random.choice(height*width, k, replace=False)
    center = result[center_point, :]
    distance = [[] for i in range(k)]

    #迭代
    for i in range(n):
        for j in range(k):
            distance[j] = np.sqrt(np.sum(np.square(result - np.array(center[j])), axis=1))
        result[:, 3] = np.argmin(np.array(distance), axis=0)
        for j in range(k):
            center[j] = np.mean(result[result[:, 3] == j], axis=0)
    return result
```

分割图像

有了 k 均值算法后，就可以分割图像了。

```
plt.subplot('121')
plt.imshow(img)

height = img.shape[0]
width = img.shape[1]

# 使用刚刚实现的k均值算法
result_img = kmeans(150, 5, img)
# 将最终结果变形为二维数组
result_img = result_img[:, 3].reshape(height, width)

plt.subplot('122')
plt.imshow(result_img)
plt.show()
```

最终可以看到 k 均值算法能够较好的对图像进行分割。


0
100
200
300
400
0
100
200
300
0
100
200
300
400
0
100
200
300

第八章 使用Apriori算法找出毒蘑菇的共性

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。

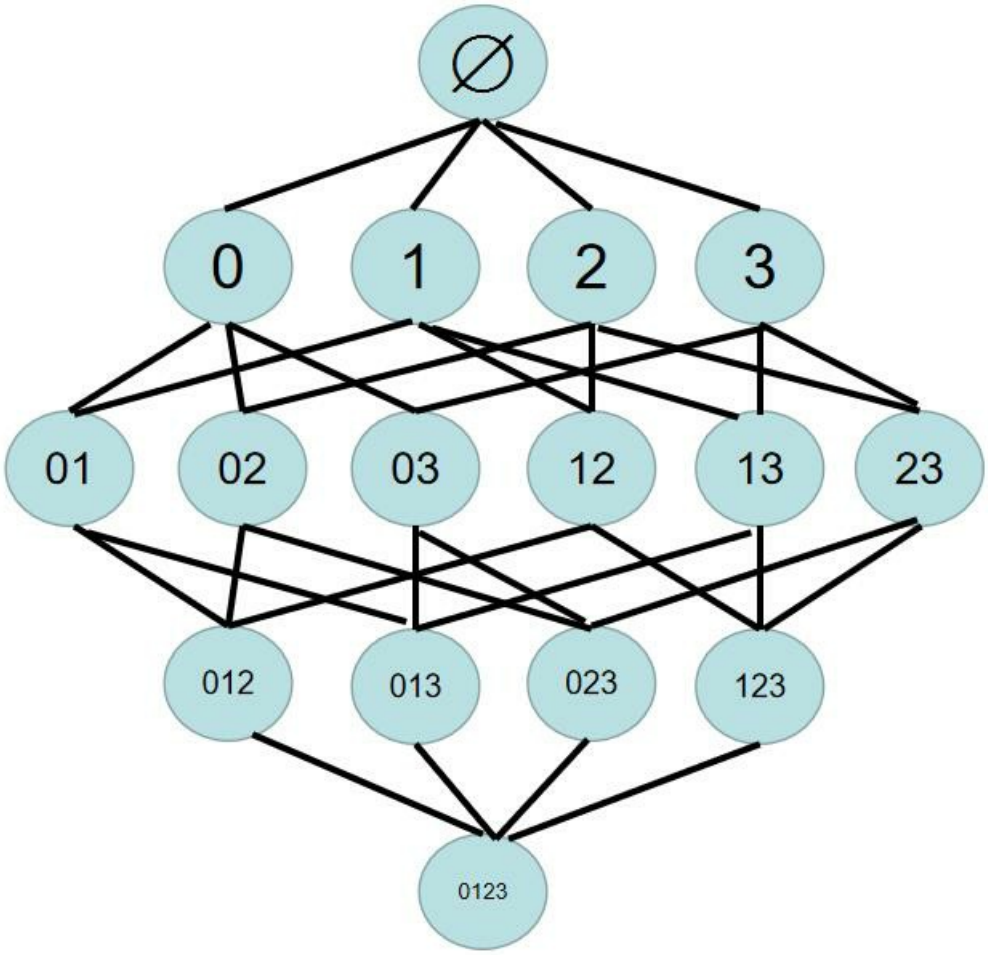
而可信度是针对关联规则而言的。例如 `{可乐}->{纸巾}` 的可信度为 `支持度({可乐，纸巾})/支持度({可乐})` ,即 `{可乐}->{纸巾}` 的可信度为 0.75 。这个 0.75 意味着对于包含可乐的所有记录，这个关联规则对其中 75% 的记录都适用。

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)：

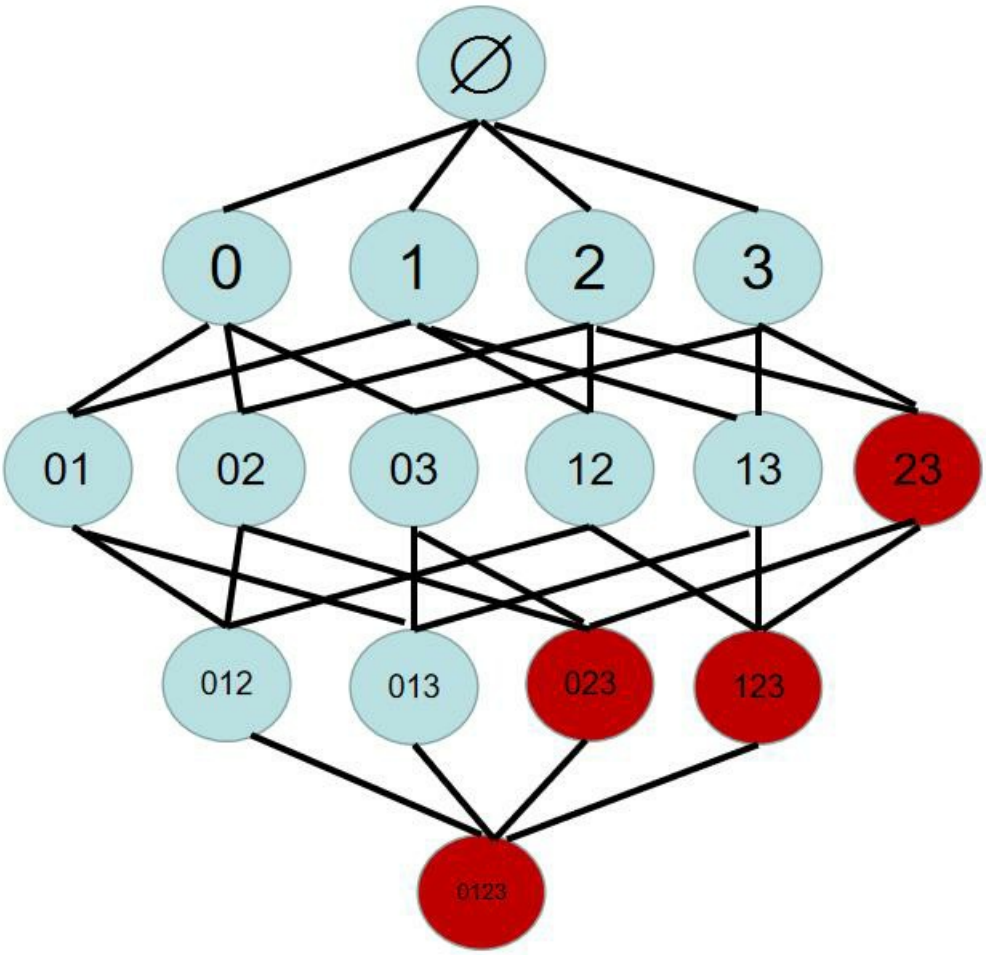


上一节说过，我们的目标是找到经常在一起购买的物品集合，并且我们通常会使用集合的支持度来度量其出现的频率。一个集合的支持度是指有多少比例的交易记录包含该集合。如何对一个给定的集合，比如 `{0, 3}` ，来计算其支持度？很显然，可以遍历每条记录并检查该记录包含 0 和 3 ，如果记录确实同时包含这两项，那么就增加总计数值。在扫描完所有数据之后，使用统计得到的总数除以总的交易记录数，就可以得到支持度。这个计算过程还仅仅是对 `{0, 3}` 这个项集而言，如果想要将海量数据中所有可能出现的项集的支持度给算出来的话，那么很用可能算一辈子也算不完。

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

想要从数据中挖掘出关联规则，就需要先能够寻找频繁项集，所以首先我们可以实现一些用于寻找频繁项集的函数。

频繁项集来源于项集，所以首先需要构建只有一个元素的项集，再构建只有两个元素的项集，...，一直到有 K 个元素的项集。因此首先需要构建只有一个元素的项集，构建的函数实现如下：

```
# 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 create_C1(dataset):
    C1 = set()
    for t in basket:
        for item in t:
            item_set = frozenset([item])
            C1.add(item_set)
    return C1
```

有了从无到有之后，接下来需要从 1 到 K。不过需要注意的是，这个时候需要排除掉支持度小于最小支持度的项集。代码实现如下：

```
# 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
```

这就完了吗？还没有，我们还需要一个能够实现构建含有 K 个元素的频繁项集的函数。实现代码如下：

```
#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
```

有了以上的函数之后，我们就可以根据生成候选频繁项集的流程，使用这些函数来实现这个功能了。代码如下：

```
#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
    # L为候选频繁项集列表，supportData为候选频繁项集的支持度
    return L, supportData
```

有了候选频繁项集和它的支持度之后，就可以开始着手挖掘关联规则了。在挖掘关联规则时，需要排除可信度比较小的关联规则，所以首先需要实现计算关联规则的可信度的功能。代码实现如下：

```
# 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
```

接下来就需要实现从频繁项集中生成关联规则的功能了，若已经忘记了算法流程，可以翻看一下上一节的内容，实现如下：

```
# 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)
```

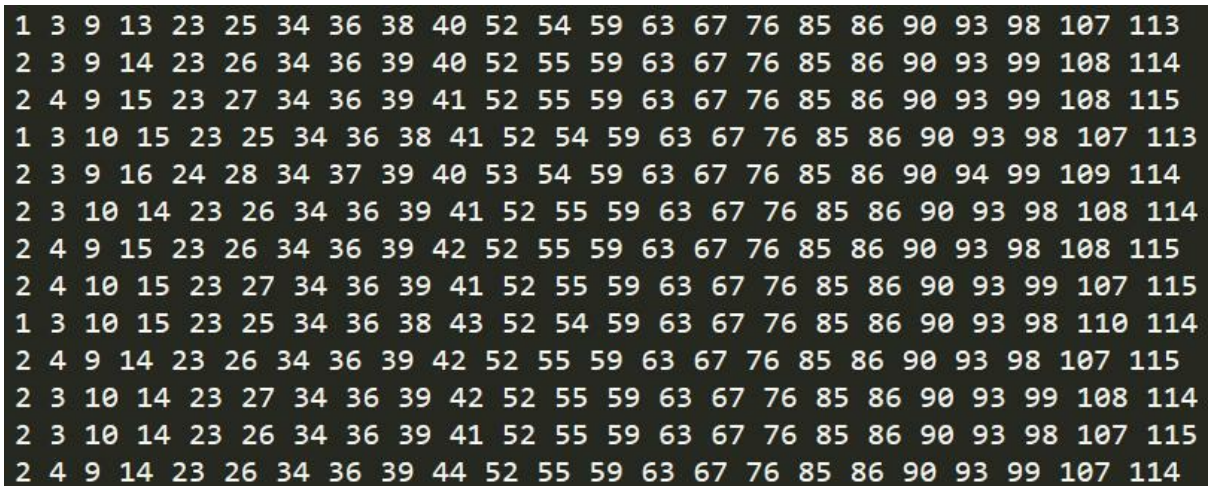
由于使用 `apriori` 函数后得到的是一个候选频繁项集的列表，所以需要遍历整个候选频繁项集的列表来挖掘关联规则，所以代码实现如下：

```
# 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
```

到这里，我们已经实现了 `Apriori` 算法和关联规则挖掘算法。接下来，将介绍如何使用刚刚实现的算法来挖掘毒蘑菇的共性特征。

8.4 发现毒蘑菇的特性

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 谷歌的网页推荐算法--PageRank

9.1 什么是PageRank

PageRank 的 Page 可以认为是网页，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。

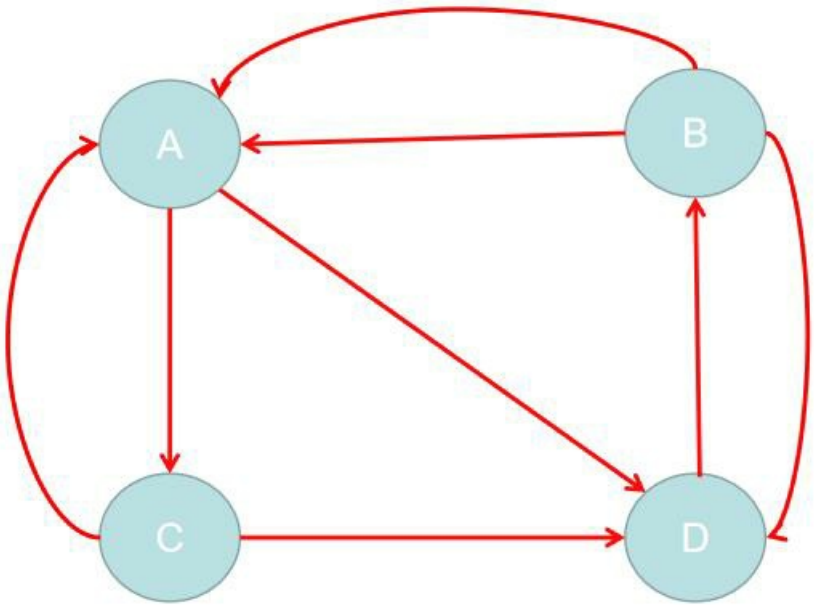
PageRank算法计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去， PageRank 就是估计这个悠闲的上网者分布在各个网页上的概率。

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成是一个有向图，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：

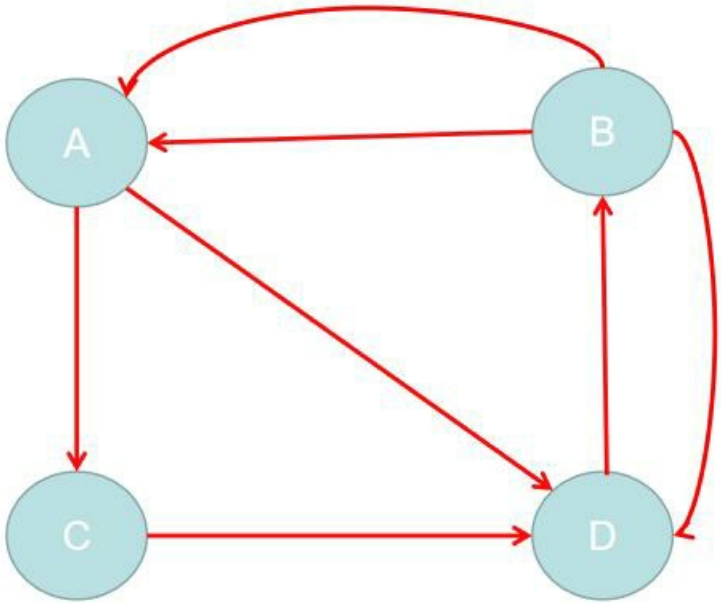
$$V_1 = MV_0 = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

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。

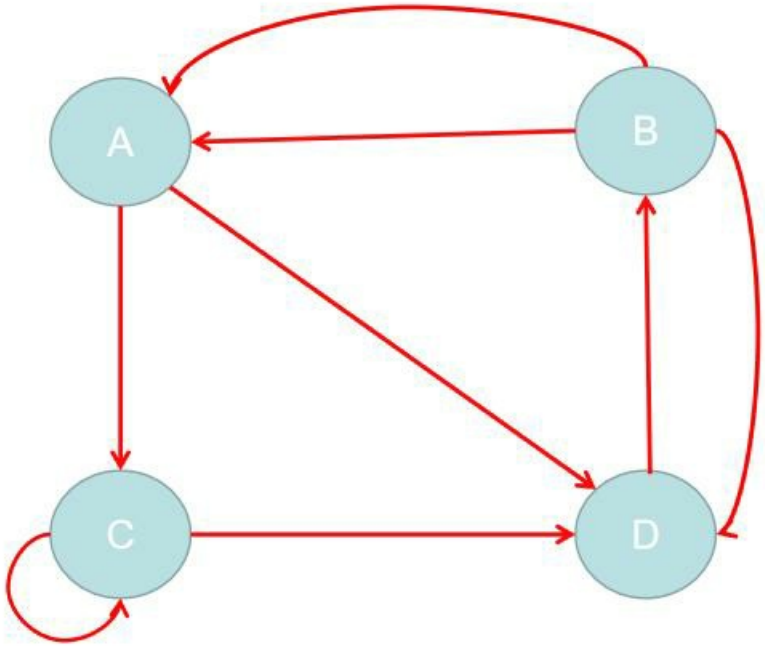


对应的转移矩阵 M 为：

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

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：



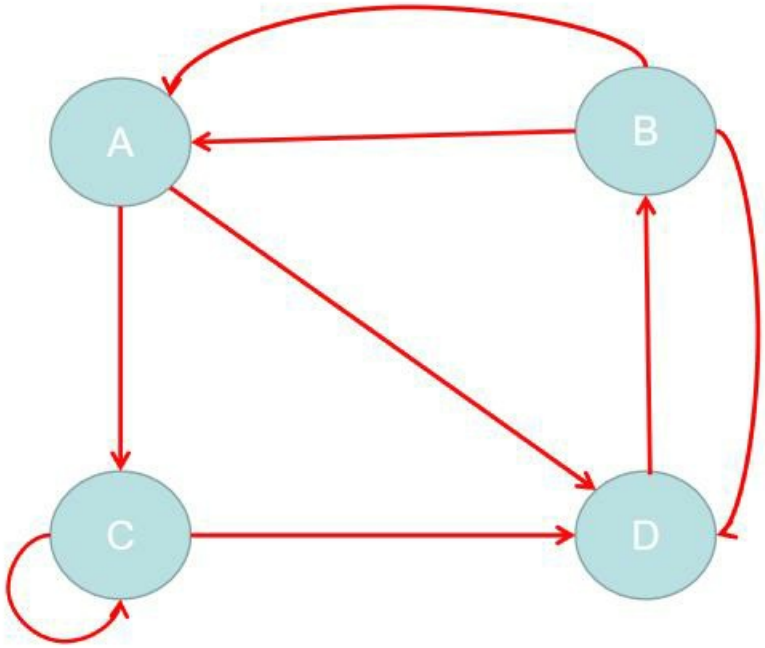
上网者跑到 C 网页后，就像陷入了漩涡，再也不能从 C 中出来，将最终导致概率分布值全部转移到 C 上来，这使得其他网页的概率分布值为 0 ，从而整个网页排名就失去了意义。

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，而是悄悄地在地址栏输入另外一个地址。像这样直接输入网址后跳转到各个网页的概率很显然是 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 打造电影推荐系统

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，向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次购买者呢（比如说汽车），这个时候怎么办？我们可以考虑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

假设现在需要为用户推荐他可能喜欢的电影，如果你是这个项目的设计者，你会以什么样的依据来对不同的用户推荐他可能喜欢的电影呢？

嗯，你可能会认为，如果我有不同用户对不同电影的评分，或者好评率等一些可以量化的数值，那么我只要找到用户对所有电影中评分最高那部电影，然后推荐给他就行了。

是的，没错！其实很多推荐系统就是这样实现的。也就是说，如果有下面这样的表格，我想你应该会像 2 号用户推荐 3 号电影。

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

但是我们的用户，不可能所有电影都看过，也就是说，在实际场景中，我们的表格中会有很多缺失值，在这里假设用 `0` 来代表缺失值，就如下表一样：

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

所以我们希望能够有一种算法，它能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。

基于矩阵分解的协同过滤算法正好能解决这个问题。假设现有的表格如下所示：

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：

- 随机初始矩阵值
- 构造损失函数，求得矩阵参数梯度
- 进行梯度下降，更新矩阵参数值
- 喜好矩阵与内容矩阵相乘得到预测评分
- 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陡的方向，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

弄清楚了基于矩阵分解的协同过滤算法的原理和算法流程之后，我们可以很容易的使用`python`，来实现算法。代码如下：

```
# -*- 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)
```

第十一章 综合实战：森林火灾数据可视化

亚马逊热带雨林位于南美洲的亚马逊平原，占地 550 万平方公里。雨林横越了9个国家：巴西、哥伦比亚、秘鲁、委内瑞拉、厄瓜多尔、玻利维亚、圭亚那、苏里南以及法国（法属圭亚那），占据了世界雨林面积的一半，占全球森林面积的 20% ，是全球最大及物种最多的热带雨林。亚马逊雨林被人们称为“地球之肺”和“绿色心脏”。

然而就在前不久，巴西爆发了前所未有的大火，并在 2019 年 8 月份愈演愈烈。今年迄今已发生超过 74000 起火灾，这是该国国家空间研究所记录的最多火灾数量。与 2018 年同一时期的火灾数量相比，这一数字大约增加了 80％ 。其中超过一半的火灾发生在亚马逊地区。

可想而知，如何对森林火灾的大量历史数据进行分析与挖掘，总结出森林发生火灾的规律，对于防止森林或者是雨林发生火灾是非常有意义的一件事情。

在这里已经有一份巴西近 20 年来亚马逊热带雨林的一些数据，我们的目标是对这些数据进行理解、分析、并对其进行可视化，来验证我们对于森林火灾的一些猜测和认知。

11.1 亚马逊雨林数据初窥

拿到数据的第一步，肯定是先看看数据中有多少个字段，每个字段代表什么意思。这份数据是一个 csv 文件，所以可以使用 pandas 来读取数据。

```
import pandas as pd

# 读取csv文件
data = pd.read_csv('./amazon.csv')

# 查看数据的前5行
data.head(5)
```

	year	state	month	number	date
0	1998	Acre	Janeiro	0.0	1998-01-01
1	1999	Acre	Janeiro	0.0	1999-01-01
2	2000	Acre	Janeiro	0.0	2000-01-01
3	2001	Acre	Janeiro	0.0	2001-01-01
4	2002	Acre	Janeiro	0.0	2002-01-01

可以看出，每一行数据表示当天在特定区域内发生火灾的次数。嗯，前 5 行数据中都没有发生过火灾，看上去还不错。

接下来，看看火灾发生次数的简单统计信息。

```
data['number'].describe()
```

```
count    6454.000000
mean      108.293163
std       190.812242
min         0.000000
25%         3.000000
50%        24.000000
75%       113.000000
max       998.000000
Name: number, dtype: float64
```

哇，从 98 年到 19 年期间，亚马逊森林总共发生了 6454 起火灾！而且平均每天发生了 108 起火灾！

这样的统计对于分析来说，粒度还是有点粗，我们不妨对火灾发生的次数进行分组统计。比如根据年、月和地区来分组，那么在对月份分组之前，可以先将数据中的月份进行简化，然后再进行分组。

```
# 将month中的月份改成缩写
data['month'].replace(to_replace = 'Janeiro', value = 'Jan', inplace = True)
```

```
data['month'].replace(to_replace = 'Fevereiro', value = 'Feb', inplace = True)
data['month'].replace(to_replace = 'Março', value = 'Mar', inplace = True)
data['month'].replace(to_replace = 'Abril', value = 'Apr', inplace = True)
data['month'].replace(to_replace = 'Maio', value = 'May', inplace = True)
data['month'].replace(to_replace = 'Junho', value = 'Jun', inplace = True)
data['month'].replace(to_replace = 'Julho', value = 'Jul', inplace = True)
data['month'].replace(to_replace = 'Agosto', value = 'Aug', inplace = True)
data['month'].replace(to_replace = 'Setembro', value = 'Sep', inplace = True)
data['month'].replace(to_replace = 'Outubro', value = 'Oct', inplace = True)
data['month'].replace(to_replace = 'Novembro', value = 'Nov', inplace = True)
data['month'].replace(to_replace = 'Dezembro', value = 'Dec', inplace = True)

# 分组统计
year_mo_state = data.groupby(by = ['year','state', 'month']).sum().reset_index()

year_mo_state
```

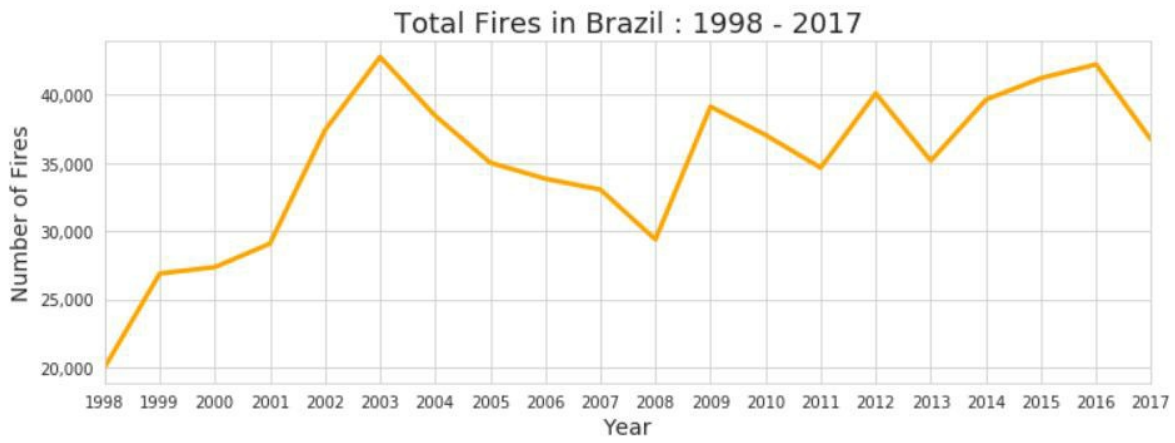
	year	state	month	number
0	1998	Acre	Apr	0.000
1	1998	Acre	Aug	130.000
2	1998	Acre	Dec	7.000
3	1998	Acre	Feb	0.000
4	1998	Acre	Jan	0.000
5	1998	Acre	Jul	37.000
6	1998	Acre	Jun	3.000
7	1998	Acre	Mar	0.000
8	1998	Acre	May	0.000
9	1998	Acre	Nov	0.000
10	1998	Acre	Oct	44.000
11	1998	Acre	Sep	509.000
12	1998	Alagoas	Apr	0.000
13	1998	Alagoas	Aug	1.000
14	1998	Alagoas	Dec	32.000
15	1998	Alagoas	Feb	0.000
16	1998	Alagoas	Jan	0.000

现在已经统计出了，某年某月某个地区发生火灾的次数。但是这样的一份不完全的报表(由于篇幅原因，这里只截出了部分统计结果)放在你的面前，你可能并不会有什么感觉，因为人类喜欢的是图，而不是一堆数字。所以下一节，将带你画出有趣的图表，让你对这份数据有一份更加直观的认识。

11.2 使用图表来探索亚马逊雨林

现在我们不妨使用 python 代码画出每年亚马逊雨林火灾发生次数的折线图。

```
# 导入matplotlib中的MaxNLocator和FuncFormatter
from matplotlib.pyplot import MaxNLocator, FuncFormatter

# 设置折线图的大小
plt.figure(figsize=(12,4))

# 绘制折线图，横轴为年份，纵轴为火灾发生次数
ax = sns.lineplot(x = 'year', y = 'number', data = year_mo_state, estimator = 'sum', color = 'orange', lw = 3
,
                  err_style = None)

# 设置折线图的标题
plt.title('Total Fires in Brazil : 1998 - 2017', fontsize = 18)
# 设置横轴与纵轴的名称
plt.xlabel('Year', fontsize = 14)
plt.ylabel('Number of Fires', fontsize = 14)

# 设置刻度
ax.xaxis.set_major_locator(plt.MaxNLocator(19))
ax.set_xlim(1998, 2017)

# 设置日期格式
ax.get_yaxis().set_major_formatter(plt.FuncFormatter(lambda x, p: format(int(x), ',')))
```

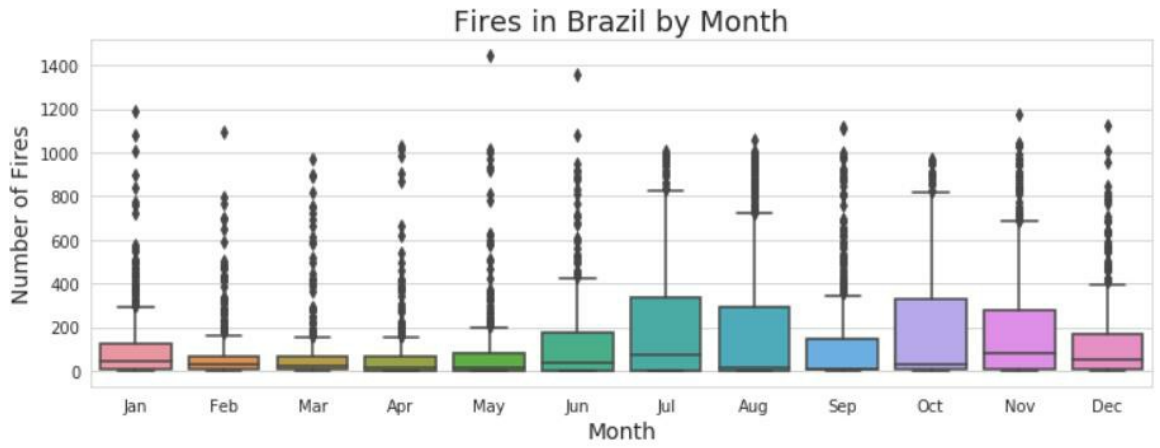


从图表可以看出，火灾在过去的 20 年中急剧增加，从 1998 年的 2 万起增加到 2017 年的 36000，将近一倍！然后，心细观察的话不难发现，火灾有不断增长的趋势，因此我们可以预计在接下来的几年里还会发生更多的火灾！

看完每年的火灾发生次数，我们再来看看每月的火灾发生次数，看看发生火灾与月份有没有什么关系。

```
# 设置折线图的大小
plt.figure(figsize=(12,4))

# 绘制盒图，横轴是月份，纵轴是火灾发生次数
sns.boxplot(x = 'month', order = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep','Oct', 'Nov',
```

```
'Dec'],
            y = 'number', data = year_mo_state)

# 设置盒的标题和横纵轴的名称
plt.title('Fires in Brazil by Month', fontsize = 18)
plt.xlabel('Month', fontsize = 14)
plt.ylabel('Number of Fires', fontsize = 14)
```



该盒图仿佛暗示着，每年的下半年，火灾发生的概率更高，尤其是从夏天到秋天这段时间。这也是符合常理的，毕竟气候干燥，容易发生火灾。

那么火灾的发生与区域有关吗？我们同样可以统计一下。

```
# 统计火灾发生次数最高的10个区域
data.groupby(by = 'state')['number'].sum().sort_values(ascending = False).head(10)
```

```
state
Mato Grosso     96246.028
Paraiba         52435.918
Sao Paulo       51121.198
Rio             45160.865
Bahia           44746.226
Piau            37803.747
Goias           37695.520
Minas Gerais    37475.258
Tocantins       33707.885
Amazonas        30650.129
Name: number, dtype: float64
```

可以看到，火灾重灾区的第二名和第三名非常接近，但是第一名比第二名高出了几乎 400000 次！嗯，我们可以网上搜索一下巴西的地图，看看这些重灾区的位置。

第一名重灾区在巴西的中部，如果发生火灾，后果可想而知。为什么 Mato Grosso 会排在榜首呢？那是因为 Mato Grosso 是巴西第三大州。这个州人口占巴西总人口的比重很小，约为 1.5 %，但农业产业非常强大。

过去，Mato Grosso 是巴西最大的二氧化碳排放州之一，这是由于其强大的农业经济驱动下的森林火灾和森林砍伐所致。

11.3 亚马逊雨林地图可视化

在上一节中，已经知道了每个区域在这将近 20 年里总共发生过多少起火灾。那么我们能不能更加直观的感受一下每年每个区域的火灾次数的变化情况呢？此时你可能会想到这样的代码：

```
data[['year','state','number']].groupby(['year','state']).number.sum()
```

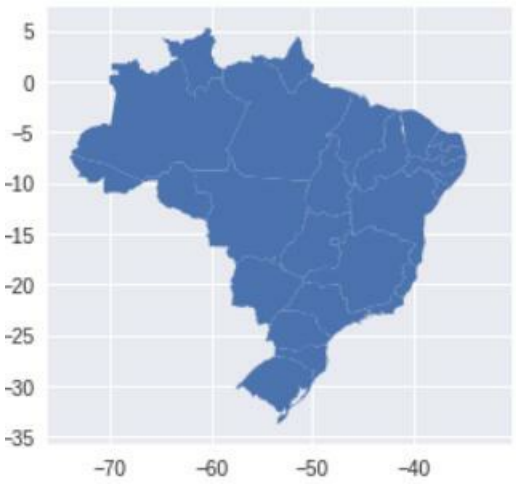
然后就会看到这样的结果。

```
year  state
1998  Acre               730.000
      Alagoas             86.000
      Amapa              278.000
      Amazonas           946.000
      Bahia             1224.687
                           ...
2017  Roraima           1101.000
      Santa Catarina    2354.000
      Sao Paulo         2540.868
      Sergipe             75.000
      Tocantins         1378.959
Name: number, Length: 460, dtype: float64
```

没错，我们需要的数据就是这些，但是我们是感官动物，看到一堆数字时并不觉得这堆数字有多棒。所以我们接下来就尝试将这些数据进行可视化，让我们能从可视化结果中看出每年这些区域的火灾次数的变化情况。

首先，我们需要一张巴西的地图，这张地图在网络上有现成的，可以从 https://geodata.lib.berkeley.edu/catalog/stanford-ys298mq8577 获取。下载好地图的压缩包后，可以将后缀名为 .shp 的文件解压出来，然后使用如下代码就可以将地图画出来。

```
# 加载geopandas
import geopandas as gp

# 读取解压好的.shp文件
geo_data = gp.read_file('./geo_brazil_data.shp')
# .shp文件中的nome字段表示state字段，所以需要字段重命名
geo_data = geo_data.rename(columns = {'nome': 'state'})
# 画图
geo_data.plot()
```

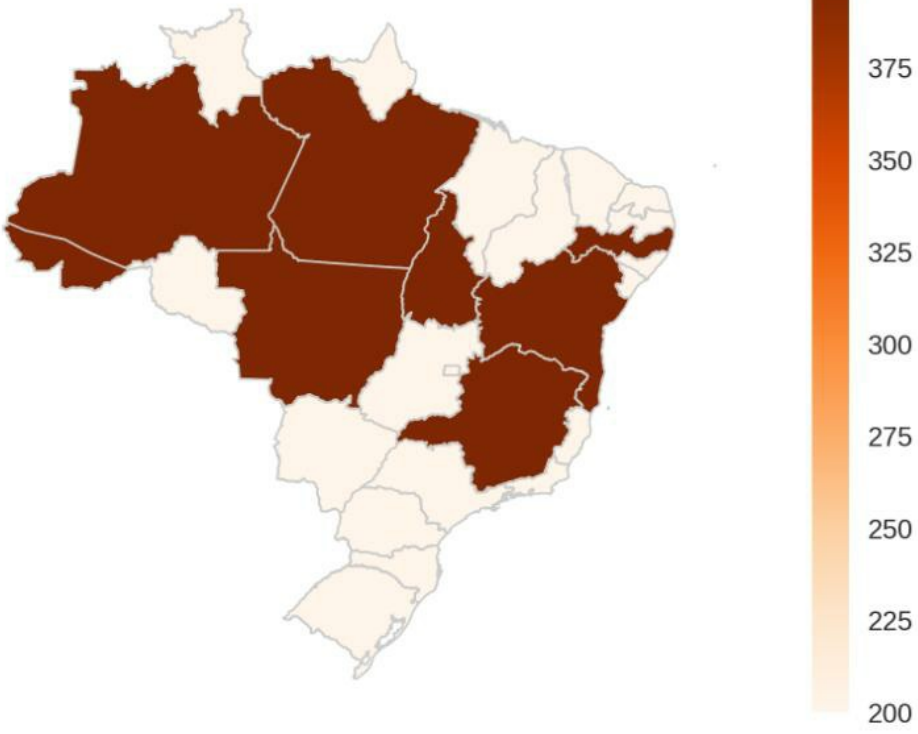


嗯，看上去是不是有点样子了？不过这还不算什么，因为这仅仅是巴西的地图轮廓而已。我们的目标是每年的火灾变化情况反应到这张地图上。所以可以编写如下代码来实现该功能：

```
# 每年每个区域的火灾次数统计
ani_d2 = data[['year','state','number']].groupby(['year','state']).number.sum().reset_index()

# 构建透视表
ani_d2_pivot = ani_d2.pivot_table(values='number', index=['state'], columns='year').reset_index()
ani_d2_final = geo_data.set_index('state').join(ani_d2_pivot.set_index('state'))

# 火灾次数的最小和最大值
vmin, vmax = 200, 400

for year in ani_d2_final.columns[1:]:
    # 创建图
    fig = ani_d2_final.plot(column=year, cmap='Oranges', figsize=(8,5), linewidth=0.8, edgecolor='0.8', vmin=
vmin, vmax=vmax, legend=True, norm=plt.Normalize(vmin=vmin, vmax=vmax))

    # 关闭轴
    fig.axis('off')

    # 设置图的标题
    fig.set_title('Fire In Brazil', fontdict={'fontsize': '15', 'fontweight' : '3'})

    # 图的注释
    fig.annotate(year, xy=(0.1, .225), xycoords='figure fraction', horizontalalignment='left', verticalalignm
ent='top',fontsize=20)
    chart = fig.get_figure()
    # 按年份保存图片
    chart.savefig(str(year) +'_fire.png', dpi=300)
    plt.close()
```

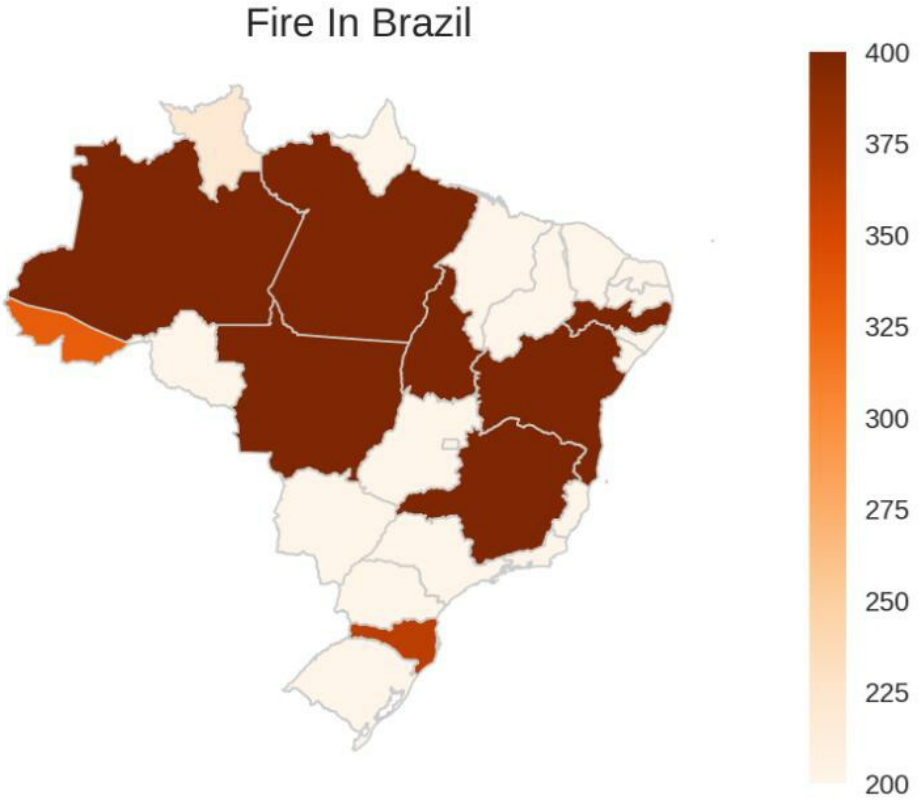
当整段代码执行完毕后，我们会在当前目录下看到 20 张 .png 图片，图片如下：

Fire In Brazil



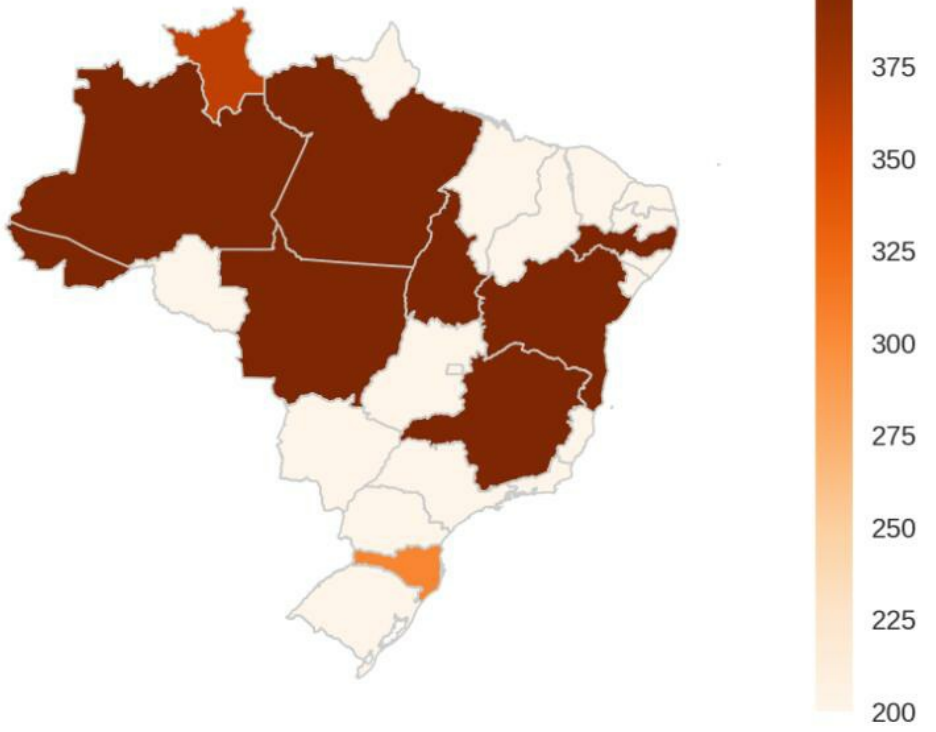
1998

Fire In Brazil



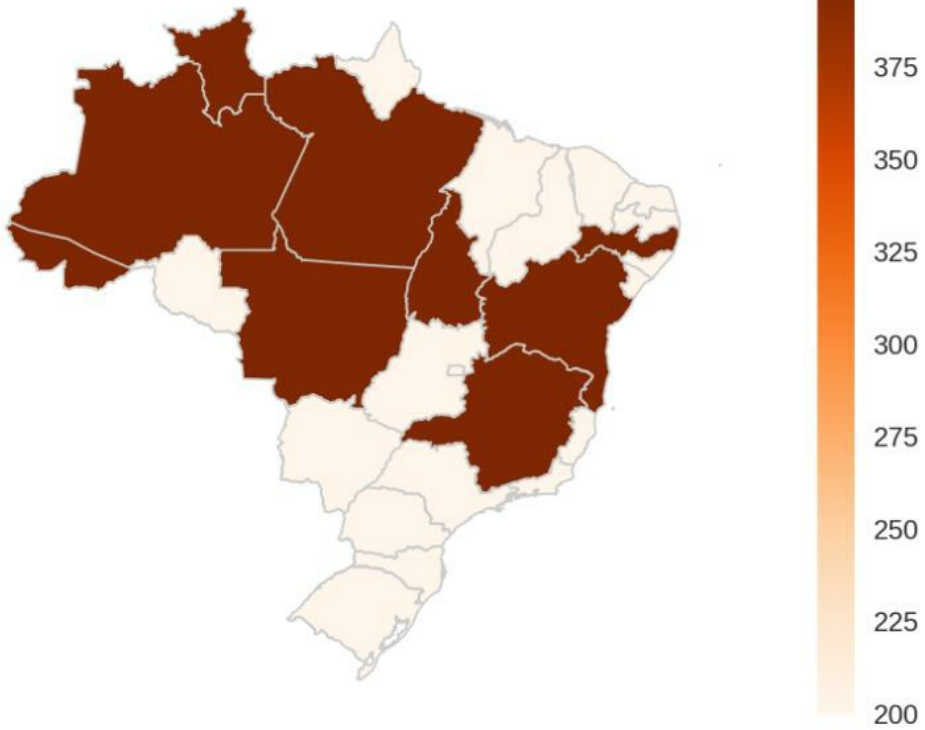
1999

Fire In Brazil



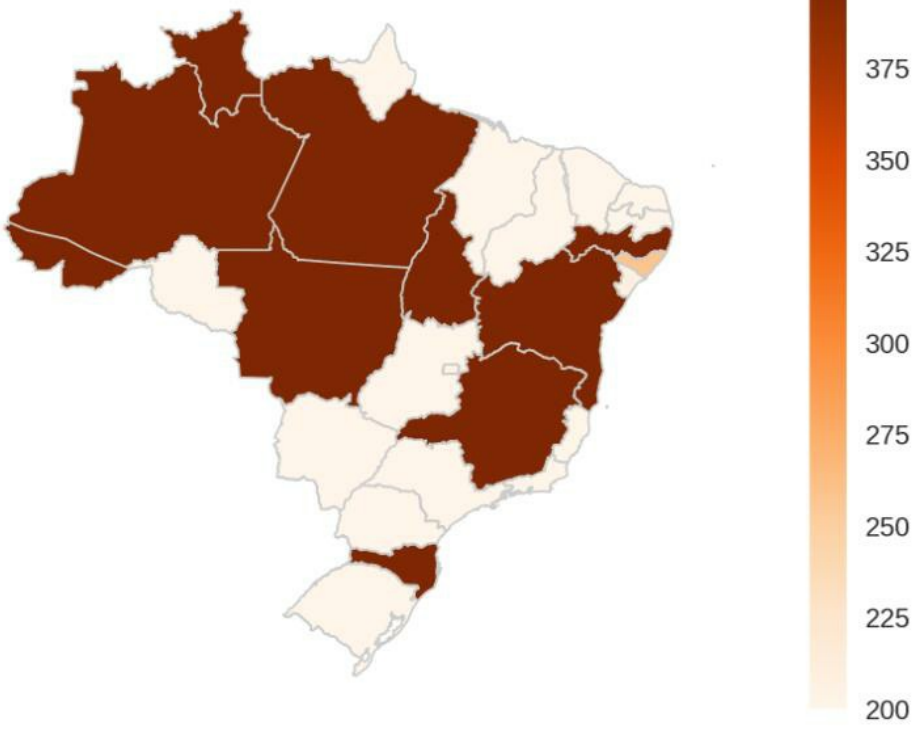
2000

Fire In Brazil



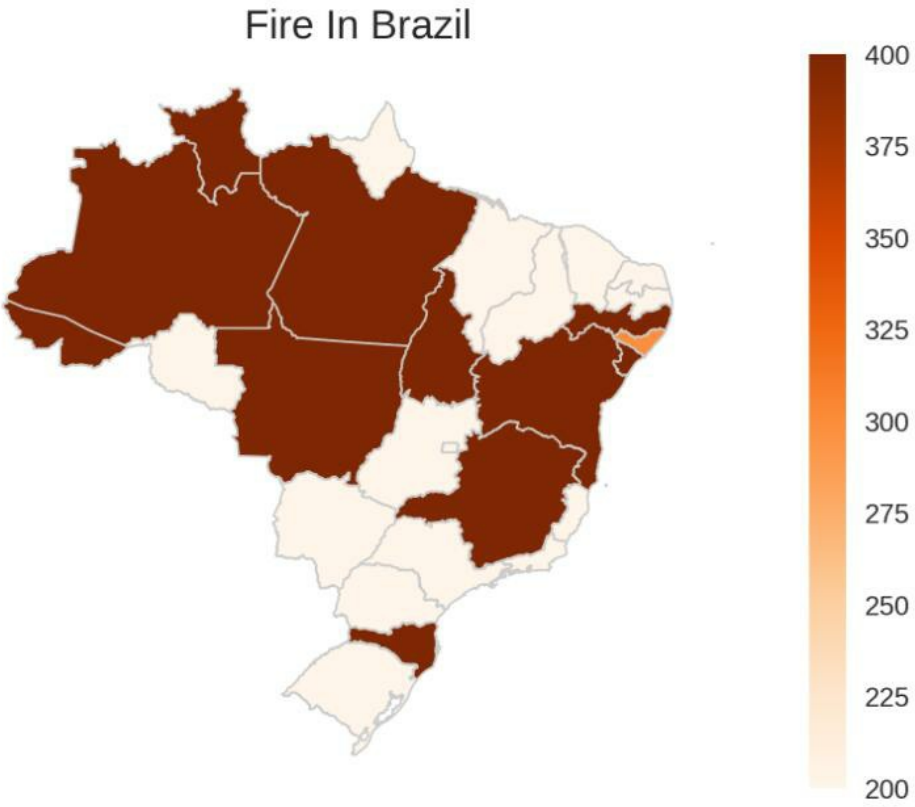
2001

Fire In Brazil



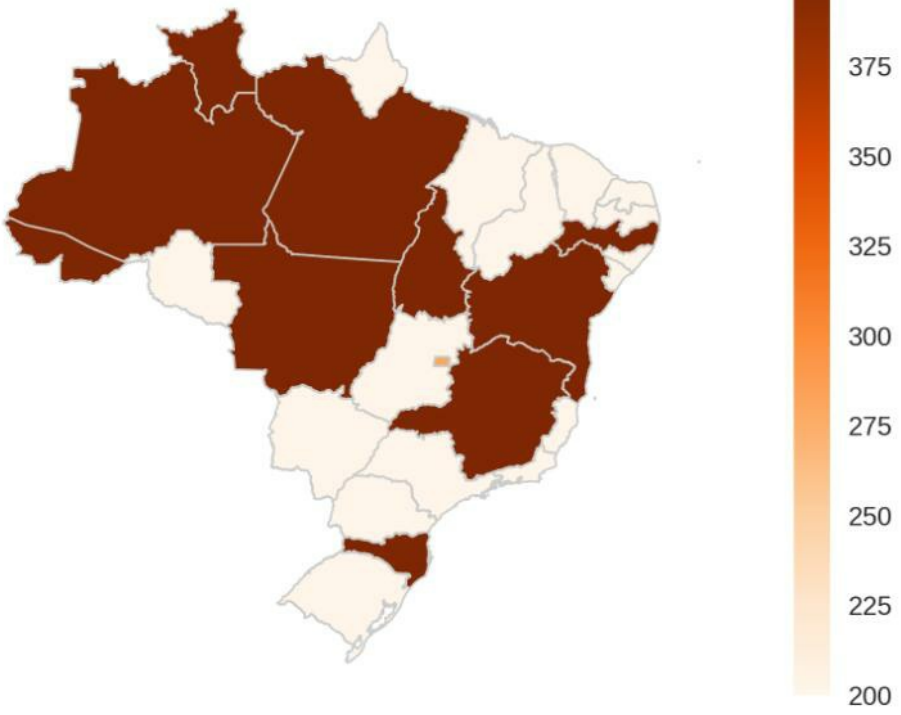
2002

Fire In Brazil



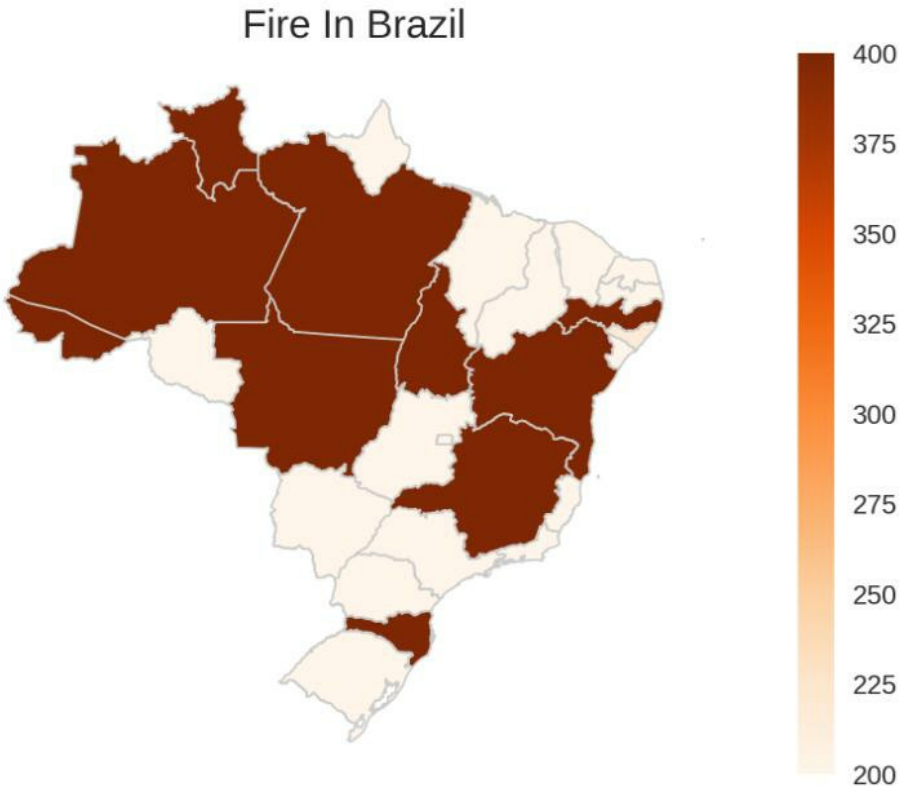
2003

Fire In Brazil



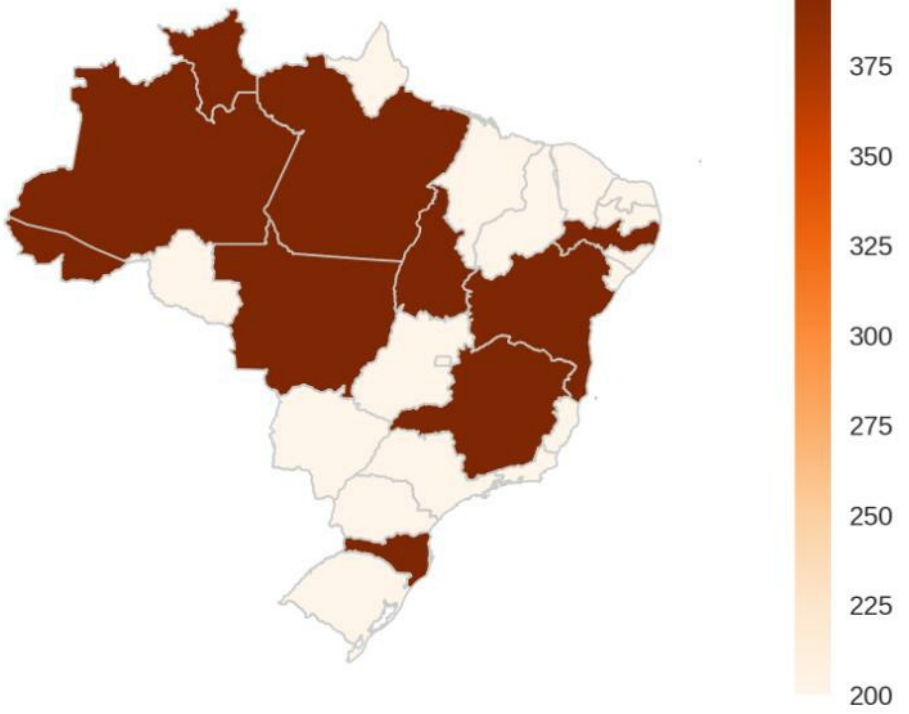
2004

Fire In Brazil



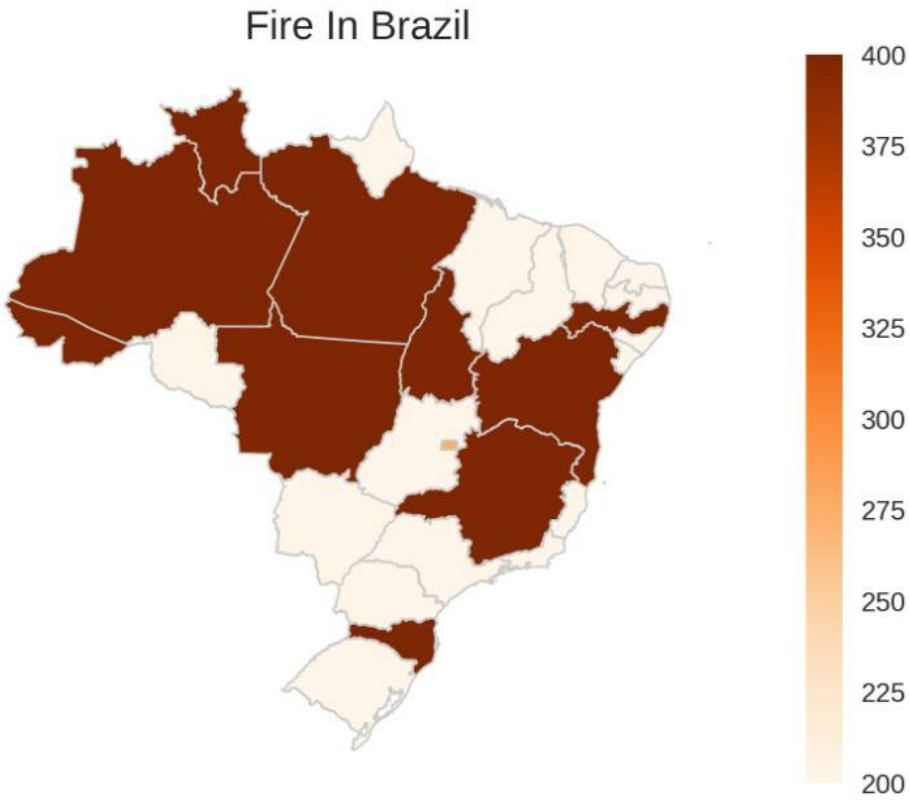
2005

Fire In Brazil



2006

Fire In Brazil



2007

Fire In Brazil

400
375
350
325
300
275
250
225
200

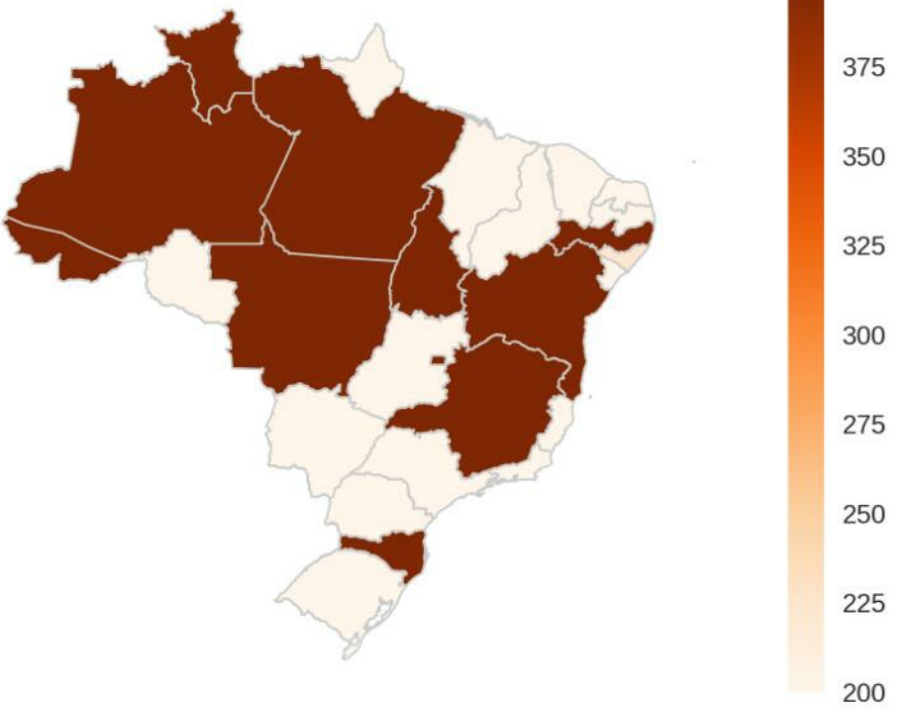
2008

Fire In Brazil

400
375
350
325
300
275
250
225
200

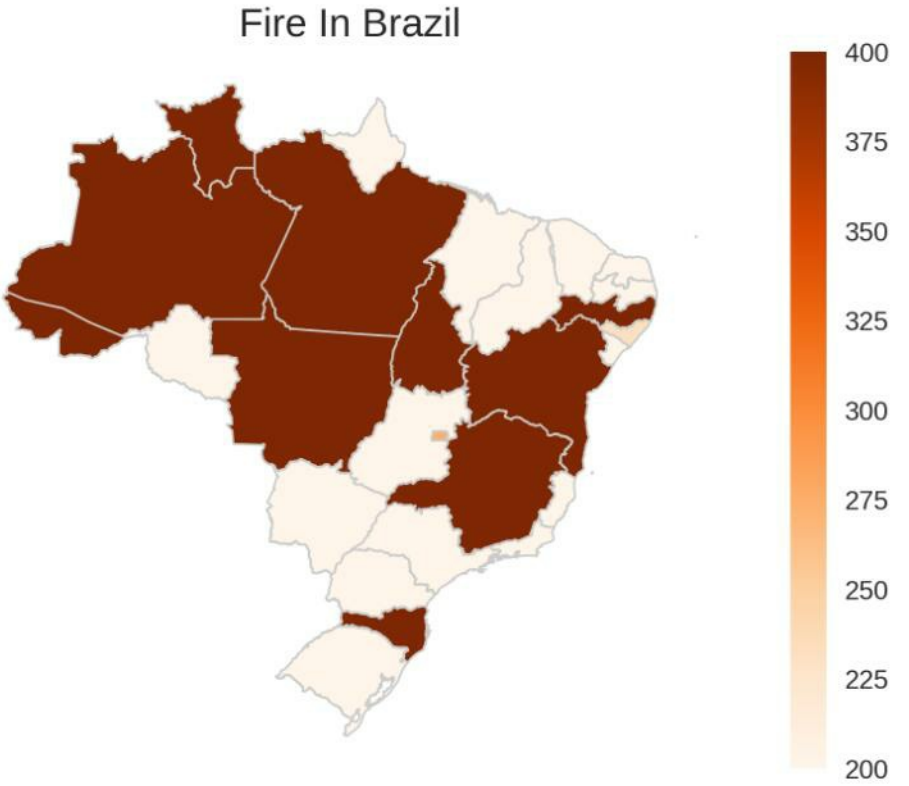
2009

Fire In Brazil



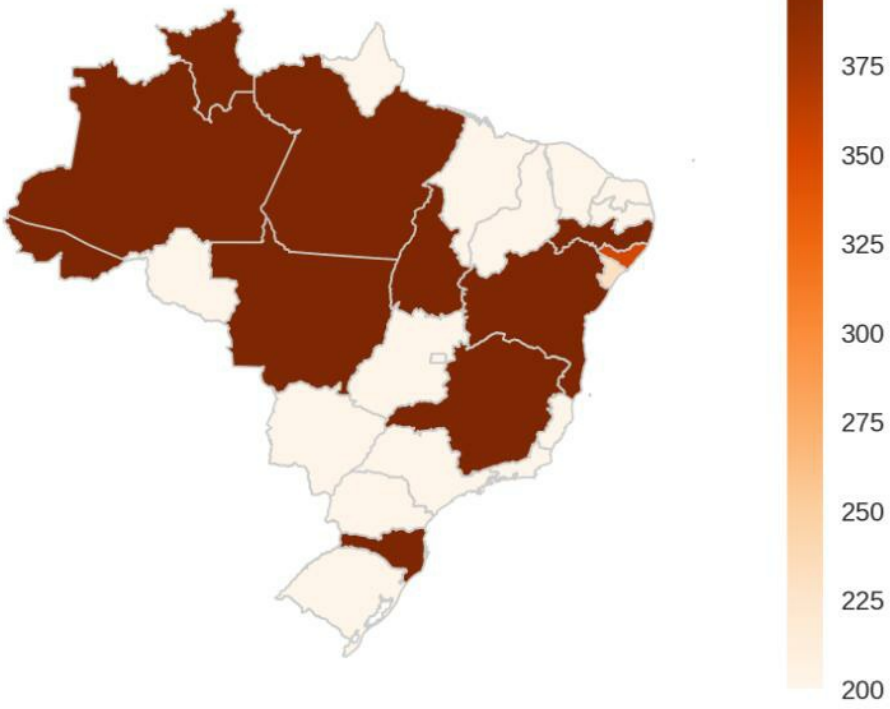
2010

Fire In Brazil



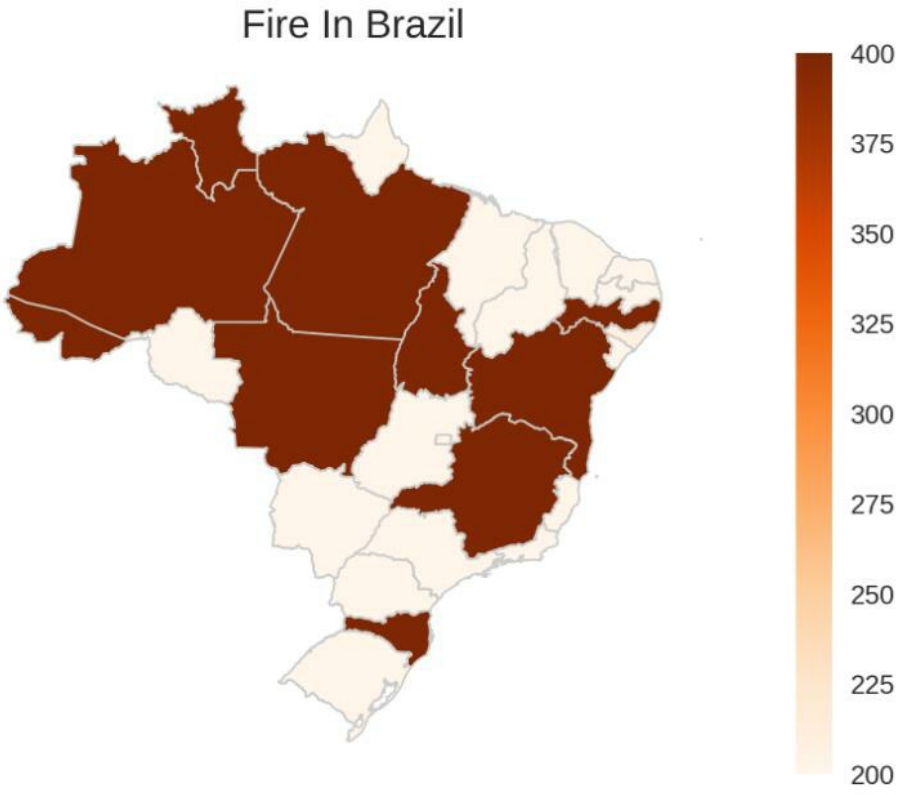
2011

Fire In Brazil



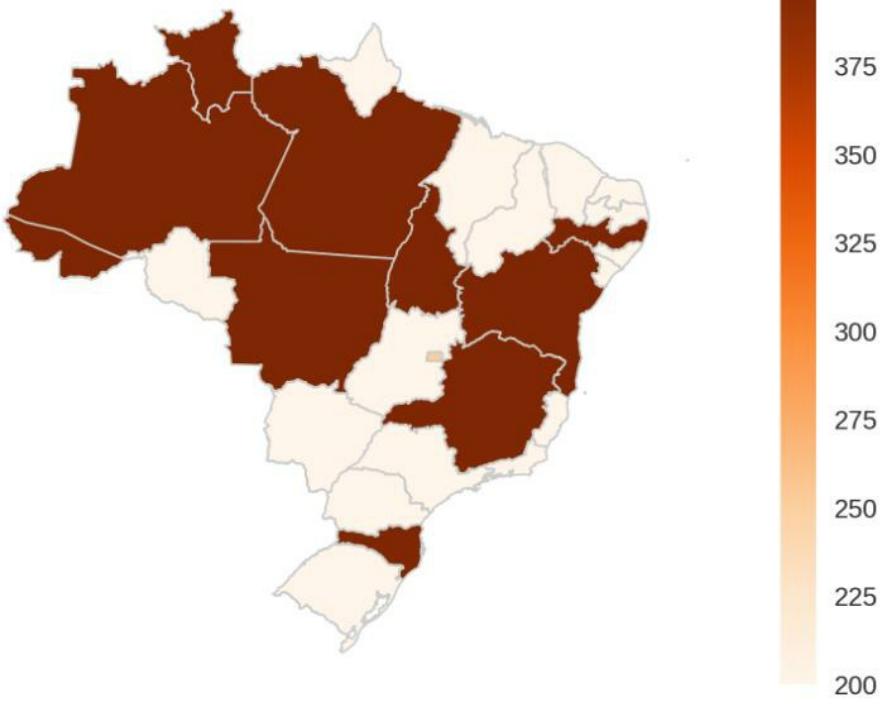
2012

Fire In Brazil



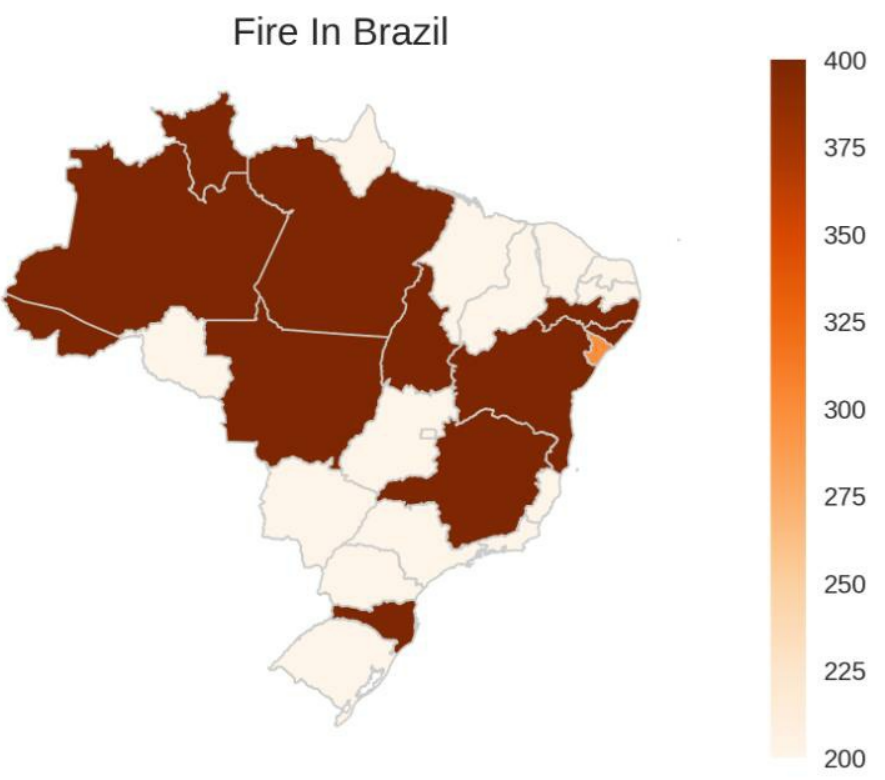
2013

Fire In Brazil



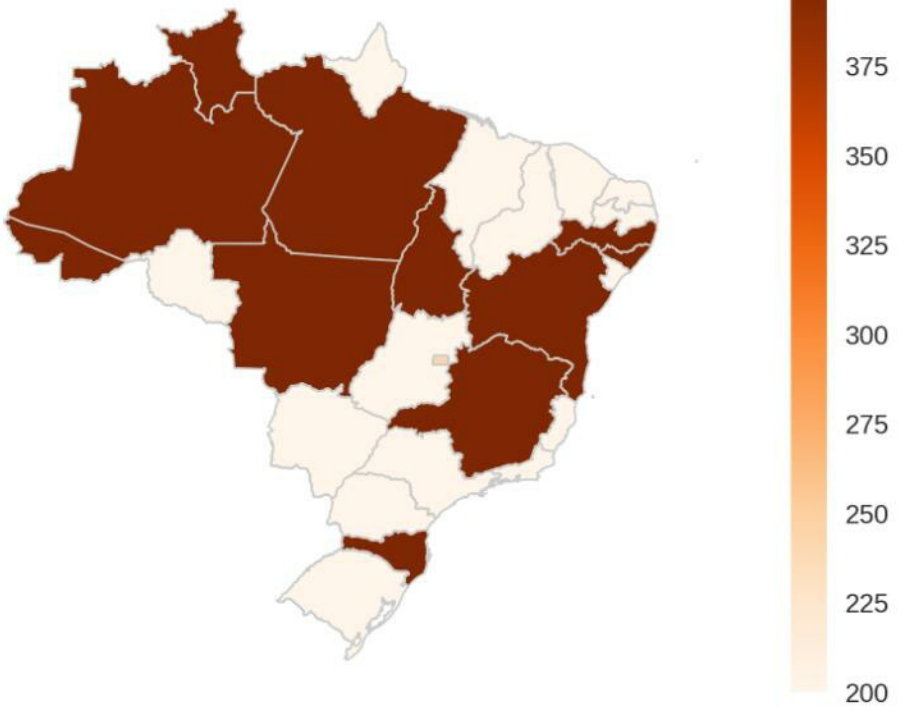
2014

Fire In Brazil



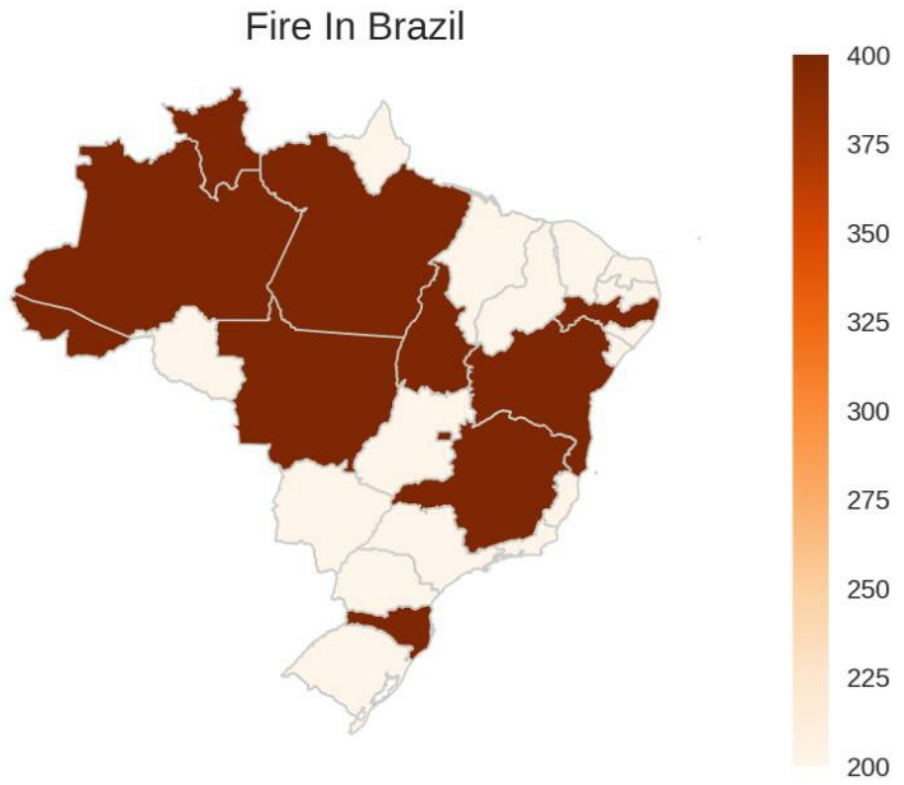
2015

Fire In Brazil



2016

Fire In Brazil



2017

可以看到，火灾的重灾区的火灾发生次数比较稳定。不过如果将这些图片合成为一张 gif 动图，可视化的效果就更加明显。那怎样将这图片转换为动图，就作为一个练习留给你练手了。

第十二章 综合实战：信用卡欺诈检测

信用卡又叫贷记卡，是由商业银行或信用卡公司对信用合格的消费者发行的信用证明。持有信用卡的消费者可以到特约商业服务部门购物或消费，再由银行同商户和持卡人进行结算，持卡人可以在规定额度内透支。

然而林子大了什么鸟都会有，有些不法分子会尝试各种手段来使用信用卡进行欺诈交易。例如本章的数据集就是欧洲银行联盟所提供的包含 2013 年 9 月欧洲持卡人通过信用卡进行的交易数据。此数据集显示两天内发生的交易。我们希望能够通过对这些数据分析、挖掘与建模，能够实现信用卡欺诈行为的检测。

12.1 了解数据

和上一章一样，我们要养成一个习惯，当拿到数据时，一定要先看看数据大概是个什么样子，有多少条记录，有多少个字段等。所以我们先来看一下数据。

```
import pandas as pd
transactions = pd.read_csv('../input/creditcard.csv')
transactions.shape
```

```
(284807, 31)
```

嗯，总共有 28 万多条信用卡交易记录，31 个特征。我们来看看大概长什么样。

```
# 随机采样5条数据
transactions.sample(5)
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
53338	45928.0	-2.080004	-0.570054	-1.206046	-0.626769	1.378741	3.721256	0.234454	0.854927	-1.759022	0.76
84760	60450.0	-3.470703	0.944028	1.046270	1.094966	-1.609649	0.363625	-0.671701	1.193442	0.413241	1.14
275945	166808.0	1.966624	-0.324929	-0.201952	0.512996	-0.684787	-0.507280	-0.527592	-0.110071	1.279241	-0.2
261764	160171.0	-0.672205	1.607262	-1.412044	-0.309166	0.299680	-0.819927	0.266643	0.691532	-0.586615	-0.9
214074	139521.0	1.950791	-0.487006	-0.377220	0.316805	-0.556397	0.012877	-0.733409	0.141586	1.257405	-0.0

特征太多，截图一次性截不全，那么可以用 info 函数来看看整个数据集的特征名称与其对应的数据类型。

```
transactions.info()
```

```
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
Time      284807 non-null float64
V1        284807 non-null float64
V2        284807 non-null float64
V3        284807 non-null float64
V4        284807 non-null float64
V5        284807 non-null float64
V6        284807 non-null float64
V7        284807 non-null float64
V8        284807 non-null float64
V9        284807 non-null float64
V10       284807 non-null float64
V11       284807 non-null float64
V12       284807 non-null float64
V13       284807 non-null float64
V14       284807 non-null float64
V15       284807 non-null float64
V16       284807 non-null float64
V17       284807 non-null float64
V18       284807 non-null float64
V19       284807 non-null float64
V20       284807 non-null float64
V21       284807 non-null float64
V22       284807 non-null float64
V23       284807 non-null float64
V24       284807 non-null float64
V25       284807 non-null float64
V26       284807 non-null float64
V27       284807 non-null float64
V28       284807 non-null float64
Amount    284807 non-null float64
Class     284807 non-null int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

不难看出数据集中的 class 字段是数据集的标签，即该交易是否为欺诈交易，1表示为敲诈交易，0 表示为正常交易。Amount 字段表示该笔交易的金额，Time 字段表示该交易发生的时间。然而其他的字段都是什么 V1, V2, V28 什么的，让我们变成了丈二的和尚摸不着头脑。这种字段我们通常称为匿名字段，为什么会有匿名字段呢？是由于该数据集是信用卡相关的数据集，里面可能会有一些客户的敏感信息，所以对原始数据进行了一些加工，来实现脱敏。

接下来，我们来看一看该数据集中是正常交易比较多还是欺诈交易比较多。

```
transactions['Class'].value_counts()
```

```
0    284315
1       492
Name: Class, dtype: int64
```

还是比较正常的，因为欺诈行为一般还是占很小一部分的。那这样会导致什么样的问题呢？很明显，假设我写一个程序，我就一直预测交易信息是属于正常交易的，那么预测的准确率也有 95% 以上。这个准确率看上去很高，但是这样的程序你敢用吗？我猜你肯定不敢用。

所以，从刚刚的数据探索我们发现，这个数据集有两个难点，一个是恶心的匿名字段，还有一个是数据集的标签分布严重倾斜。

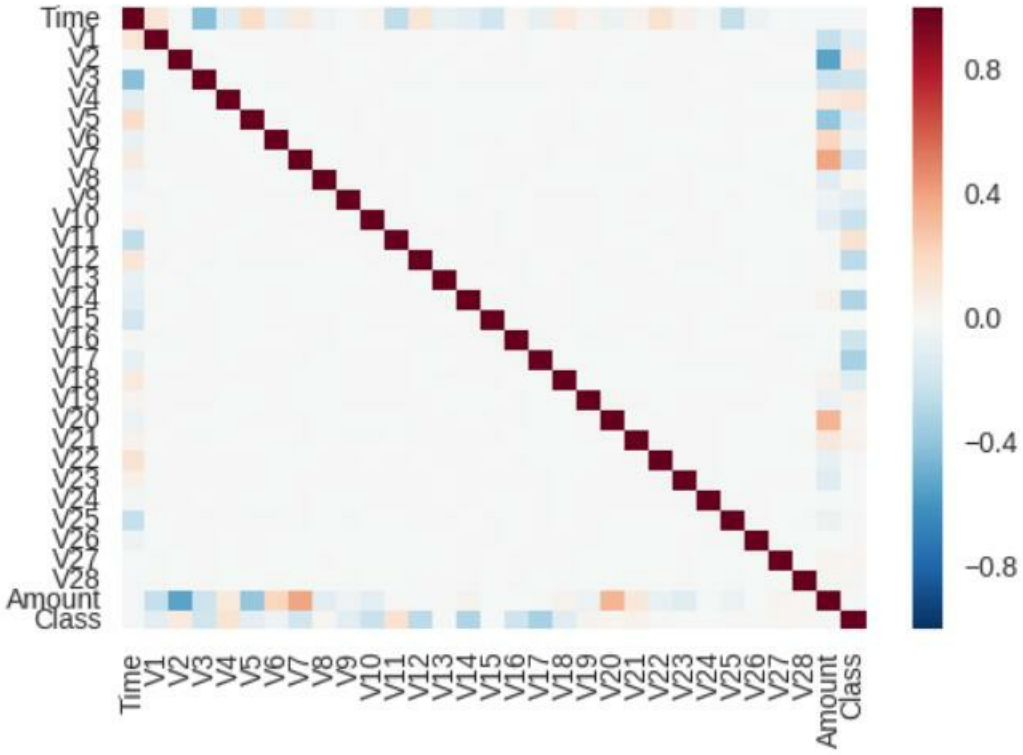
12.2 对匿名特征进行处理

首先来解决第一个问题：匿名特征。匿名特征是个很棘手的问题，因为我们并不知道特征的具体含义。所以，我们一般需要从这些特征的一些属性来分析。

我们可以尝试看看这些特征之间的协方差。协方差（Covariance）在概率论和统计学中用于衡量两个变量的总体误差，这与只表示一个变量误差的方差不同。 如果两个变量的变化趋势一致，也就是说如果其中一个大于自身的期望值，另外一个也大于自身的期望值，那么两个变量之间的协方差就是正值。 如果两个变量的变化趋势相反，即其中一个大于自身的期望值，另外一个却小于自身的期望值，那么两个变量之间的协方差就是负值。

也就是说，当两个特征之间的协方差趋近于 0 时，那么这两个特征之间可能没有太多的相关性。可以通过以下代码计算特征之间的协方差，并以热力图的方式可视化出来。

```
# 导入seaborn
import seaborn as sns

sns.heatmap(transactions.corr())
```



从热力图可以看出，匿名特征与匿名特征之间的协方差几乎趋近于 0 。那这个时候可以大胆的猜测，这些匿名特征是原始特征经过 PCA 算法转换后的 28 个主成分。因为 PCA 算法计算后的数据有一个特性，就是协方差为 0 。

PCA 算法是一种无监督的降维算法，所谓的降维就是将原始数据的特征数量减少至你所指定的特征数量。即例如原始数据中有 108 个特征，但是现在只想保留其中比较重要且比较有用的 28 个特征。那么此时就可以使用 PCA 算法来对数据进行降维。

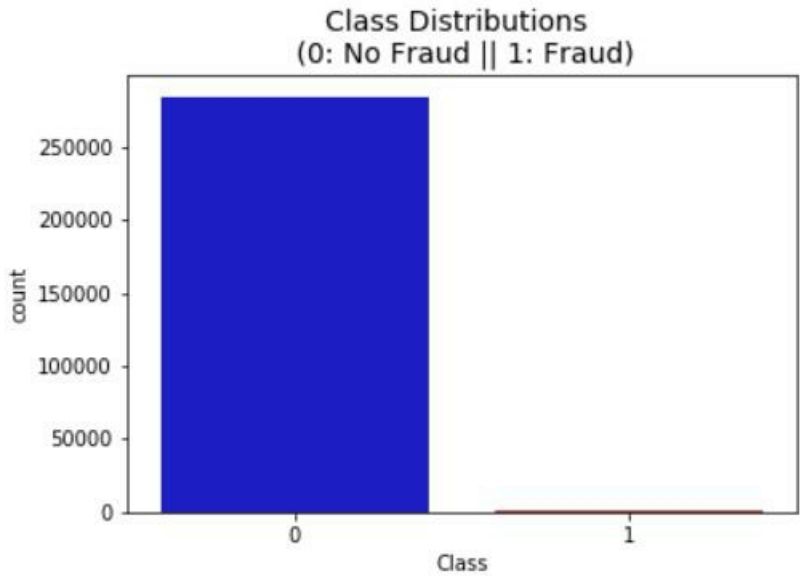
注意：PCA并不是从那 108 个特征中选择 28 个特征出来，而是经过了一系列的计算之后所产生的 28 个特征。

既然 V1 到 V28 这 28 个特征是对原始数据进行 PCA 算法处理之后的特征。所以就没太大的必要去对这些匿名特征进行特征工程了，因为这些特征已经足够独立了(没什么相关性)。

12.3 解决样本不平衡问题

在本章的第一节中已经提到过，该数据集有着非常严重的数据倾斜。

```
sns.countplot('Class', data=transactions)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=14)
```

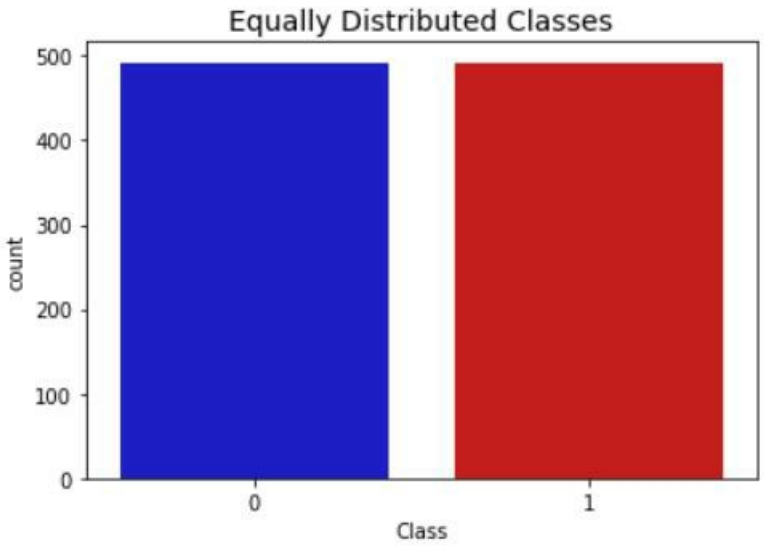


那么对于这种样本极不平衡的数据我们有两种大致的思路，一种是降采样，另一种是过采样。

降采样，顾名思义就是从标签所占比例非常高的那一部分样本中随机选取一些样本，至于选取多少个样本，就要看标签所占比例非常低的那一部分的样本有多少个了。对于该数据集来说，我们需要从标签值为 0 的样本中随机选取 492 个左右的样本出来，然后与标签值为 1 的样本组合成一个新的数据集。这就是降采样的意思。

```
# 从标签值为0的数据中随机抽取492个样本
df0 = transactions[transactions['Class'] == 0].sample(492)
# 提取出标签值为1的数据
df1 = transactions[transactions['Class'] == 1]
# 将两部分数据合并在一起
all_df = pd.concate([df0, df1], axis=0)
```

然后依葫芦画瓢的将数据的分布图可视化出来，就能看到经过降采样之后的数据分布变成了均匀分布。



但是降采样有个问题，就是丢弃了非常多的数据所提供的信息，因为我们是从 28 万条数据中随机选取的，所以无形之中就丢弃掉了大量信息！

那么有没有一种即让数据呈现出均匀分布的样子，又不会丢失信息呢？有！那就是过采样。一般来说，现在经常使用的过采样算法是 SMOTE 算法。SMOTE 算法与随机欠采样不同，SMOTE 算法会创建新的数据，以便在类之间保持相等的平衡。这是解决数据不平衡问题的另一种很棒的选择。

SMOTE 算法的大致思路是根据标签所占比例较少的那一部分数据创建一些合成点，这些合成点可以看成是数据。而这些数据中字段的值，是最近邻的思想来构建的。所以使用 SMOTE 算法来解决样本不平衡的问题，实际上就是通过使用 SMOTE 算法来生成一些新的数据，然后与原始数据合并起来，让整个数据变成均匀分布。下面是使用 SMOTE 算法实现过采样的示例代码：

```
# 导入SMOTE
from imblearn.over_sampling import SMOTE

# 使用SMOTE算法实现过采样
transactions = SMOTE().fit_sample(transactions)
```

好了，这节主要介绍了解决样本不平衡问题的两种方法，至于哪种方法的效果好，你可以自己在本章结束之后自己动手试试。至于过采样之后的数据分布图是怎样的，这里就当给你留的一个作业吧。

12.4 使用sklearn实现欺诈检测功能

前两节已经解决掉了这个数据集的两个难点，那么接下来，要做的事情就是一些非常基本的处理，来对该数据集进行建模，从而实现欺诈检测的功能了。

不过在建模之前，我们需要对 Time 和 Amount 这两个特征进行分析和处理。

首选是 Time ，可以看看 Time 的一个分布。

```
transactions['Time'].describe()
```

```
count    227845.000000
mean      94707.617670
std       47523.204111
min           0.000000
25%       54086.000000
50%       84609.000000
75%      139261.000000
max      172792.000000
Name: Time, dtype: float64
```

Time 特征的值是比较大的数，看起来像是以秒为单位的时间戳。那么这个时候我们可以将该时间戳转换成分钟和小时。

```
# 将Time转换成秒为单位的时间戳
timedelta = pd.to_timedelta(transactions['Time'], unit='s')
# 创建一个新的特征Time_min，表示分钟
transactions['Time_min'] = (timedelta.dt.components.minutes).astype(int)
# 创建一个新的特征Time_hour，表示小时
transactions['Time_hour'] = (timedelta.dt.components.hours).astype(int)
```

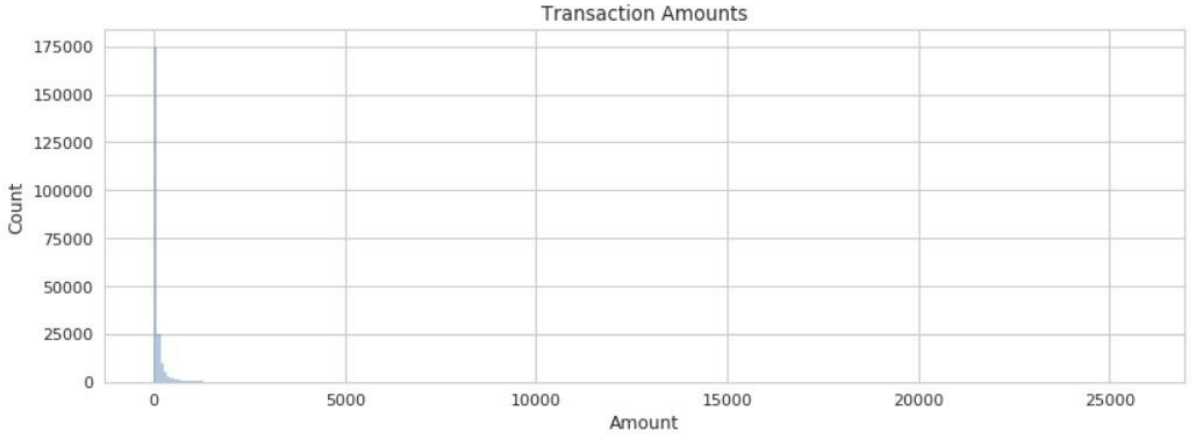
然后是 Amount 特征，同样，先看看分布。

```
transactions['Amount'].describe()
```

```
count    227845.000000
mean         88.709296
std         250.026305
min           0.000000
25%           5.550000
50%          22.000000
75%          77.890000
max       25691.160000
Name: Amount, dtype: float64
```

从分布来看，Amount 特征好像存在着严重的倾斜。我们把直方图画出来验证一下我们的猜测。

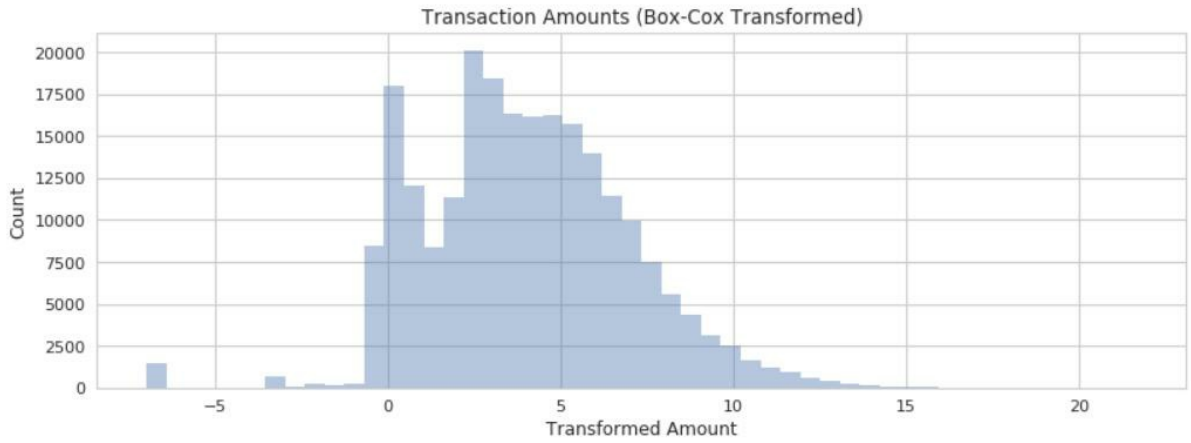
```
plt.figure(figsize=(12,4), dpi=80)
sns.distplot(transactions['Amount'], bins=300, kde=False)
plt.ylabel('Count')
plt.title('Transaction Amounts')
```



果然！大量数据的 Amount 都接近于 100 左右。一般在碰到这种倾斜严重的特征时，我们需要对它进行 log 变换，log 变换的意图就是将倾斜严重的分布，尽量让它变得更均匀。

```
transactions['Amount_log'] = np.log(transactions.Amount + 0.01)
```

然后再可视化一下，可以看出经过 log 变换之后，分布变得更加均匀了。



接下来可以使用我们的特征来构造欺诈检测模型了。构建模型很简单，使用 sklearn 提供的接口即可。

```
# 导入imblearn中的pipeline功能
from imblearn.pipeline import make_pipeline as make_pipeline_imb
# 导入上一节中提到的SMOTE功能
from imblearn.over_sampling import SMOTE

# 选用一些特征
transactions = transactions[["Time_hour","Time_min","V2","V3","V4","V9","V10","V11","V12","V14","V16","V17","
V18","V19","V27","Amount_log","Class"]]

# 构建pipeline，pipeline的意思是流水线，在这个欺诈检测的流水线中做了两件事情
# 1. SMOTE过采样
# 2. 使用决策树来进行分类，即欺诈检测
smote_pipeline = make_pipeline_imb(SMOTE(), DecisionTreeClassifier()
)
```

有了 pipeline 之后，我们就相当于已经有了一条能够检测欺诈交易的流水线了。不过这样一条流水线的效果好不好，需要使用数据来进行验证。下一节中将会向你介绍怎样验证算法的效果。

12.5 验证算法性能

上一节，我们已经学会了怎样使用 sklearn 提供的接口来构建一棵决策树模型。那这节就要来看看我们构建的模型的准确度高不高了。然而验证该模型的性能存在两个问题。一是什么样的指标能够衡量该模型的性能，二是怎样才能不偏不倚地验证算法的性能。接下来，我会一一解答。

首先是准确度的量化问题。本章一开始就提到过，该数据集不能用准确率这个指标来衡量我们的算法的性能。因为该数据集的标签是不平衡的。那什么样的指标能够衡量这种不平衡的数据呢？那就是 F1 Score !

想要弄明白 F1 Score 所代表的意义，就需要先从混淆矩阵说起。以癌症检测系统为例，癌症检测系统的输出不是有癌症就是健康，这里为了方便，就用 1 表示患有癌症，0 表示健康。假设现在拿 10000 条数据来进行测试，其中有 9978 条数据的真实类别是 0 ，系统预测的类别也是 0 ，有 2 条数据的真实类别是 1 却预测成了 0 ，有 12 条数据的真实类别是 0 但预测成了 1 ，有 8 条数据的真实类别是 1 ，预测结果也是 1 。

如果我们把这些结果组成如下矩阵，则该矩阵就成为混淆矩阵。

真实预测	**0**	**1**
0	9978	12
1	2	8

混淆矩阵中每个格子所代表的的意义也很明显，意义如下：

真实预测	**0**	**1**
0	预测 `0` 正确的数量	预测 `1` 错误的数量
1	预测 `0` 错误的数量	预测 `1` 正确的数量

如果将正确看成是 True ，错误看成是 False ， 0 看成是 Negtive ， 1 看成是 Positive 。然后将上表中的文字替换掉，混淆矩阵如下：

真实预测	**0**	**1**
0	TN	FP
1	FN	TP

因此 TN 表示真实类别是 Negtive ，预测结果也是 Negtive 的数量； FP 表示真实类别是 Negtive ，预测结果是 Positive 的数量；FN 表示真实类别是 Positive ，预测结果是 Negtive 的数量； TP 表示真实类别是 Positive ，预测结果也是 Positive 的数量。

很明显，当 FN 和 FP 都等于 0 时，模型的性能应该是最好的，因为模型并没有在预测的时候犯错误。即如下混淆矩阵：

真实预测	**0**	**1**
0	9978	0
1	0	22

所以模型分类性能越好，混淆矩阵中非对角线上的数值越小。

然后还需要明白两个概念：精准率和召回率。

精准率**(Precision)** 指的是模型预测为 Positive 时的预测准确度，其计算公式如下：

$$Precisioin = \frac{TP}{TP+FP}$$

假如癌症检测系统的混淆矩阵如下：

真实预测	**0**	**1**
0	9978	12
1	2	8

则该系统的精准率=8/(8+12)=0.4。

0.4 这个值表示癌症检测系统的预测结果中如果有 100 个人被预测成患有癌症，那么其中有 40 人是真的患有癌症。也就是说，精准率越高，那么癌症检测系统预测某人患有癌症的可信度就越高。

召回率**(Recall)** 指的是我们关注的事件发生了，并且模型预测正确了的比值，其计算公式如下：

$$Precisioin = \frac{TP}{FN+TP}$$

假如癌症检测系统的混淆矩阵如下：

真实预测	**0**	**1**
0	9978	12
1	2	8

则该系统的召回率=8/(8+2)=0.8。

从计算出的召回率可以看出，假设有 100 个患有癌症的病人使用这个系统进行癌症检测，系统能够检测出 80 人是患有癌症的。也就是说，召回率越高，那么我们感兴趣的对象成为漏网之鱼的可能性越低。

那么精准率和召回率之间存在着什么样的关系呢？举个例子，假设有这么一组数据，菱形代表 Positive ，圆形代表 Negtive 。

现在需要训练一个模型对数据进行分类，假如该模型非常简单，就是在数据上画一条线作为分类边界。模型认为边界的左边是 Negtive，右边是Positive 。如果该模型的分类边界向左或者向右移动的话，模型所对应的精准率和召回率如下图所示：

从上图可知，模型的精准率变高，召回率会变低，精准率变低，召回率会变高。

那么有没有像准确率一样值越高说明性能越好，而且能够兼顾精准率和召回率的指标呢？有！那就是F1 Score！

F1 Score 是统计学中用来衡量二分类模型精确度的一种指标。它同时兼顾了分类模型的准确率和召回率。F1 Score 可以看作是模型准确率和召回率的一种加权平均，它的最大值是 1 ，最小值是 0 。其公式如下：

$$F1 = \frac{2*precision*recall}{precision+recall}$$

- 假设模型 A 的精准率为 0.2 ，召回率为 0.7 ，那么模型 A 的 F1 Score 为 0.31111 。
- 假设模型 B 的精准率为 0.7 ，召回率为 0.2 ，那么模型 B 的 F1 Score 为 0.31111 。
- 假设模型 C 的精准率为 0.8 ，召回率为 0.7 ，那么模型 C 的 F1 Score 为 0.74667 。
- 假设模型 D 的精准率为 0.2 ，召回率为 0.3 ，那么模型 D 的 F1 Score 为 0.24。

从上述 4 个模型的各种性能可以看出，模型C的精准率和召回率都比较高，因此它的 F1 Score 也比较高。而其他模型的精准率和召回率要么都比较低，要么一个低一个高，所以它们的 F1 Score 比较低。

这也说明了只有当模型的精准率和召回率都比较高时 F1 Score 才会比较高。这也是 F1 Score 能够同时兼顾精准率和召回率的原因。

嗯，现在知道用什么指标来衡量模型性能了，那怎样才能不偏不倚地判别模型性能的好坏呢？那就是交叉验证！

一般在真实业务中，我们可能没有真正意义上的测试集，或者说不知道测试集中的数据长什么样子。那么怎样在没有测试集的情况下来验证模型好还是不好呢？这个时候就需要验证集了。

那么验证集从何而来，很明显，可以从训练集中抽取一小部分的数据作为验证集，用来验证模型的性能。

但如果仅仅是从训练集中抽取一小部分作为验证集的话，有可能会让对模型的性能有一种偏见或者误解。

比如现在要对手写数字进行识别，那么我就可能会训练一个分类模型。但可能模型对于数字 `1` 的识别准确率比较低 ，而验证集中没多少个数字为 `1` 的样本，然后用验证集测试完后得到的准确率为 0.96 。然后您可能觉得哎呀，我的模型很厉害了，但其实并不然，因为这样的验证集让您的模型的性能有了误解。那有没有更加公正的验证算法性能的方法呢？有，那就是**k-**折交叉验证！

在**K-**折交叉验证中，把原始训练数据集分割成K个不重合的■数据集，然后做K次模型训练和验证。每■次，使■ 个■数据集验证模型，并使■其它K−1个■数据集来训练模型。在这K次训练和验证中，每次■来验证模型的■数据集都不同。最后，对这K次在验证集上的性能求平均。

OK，明白了什么是 F1 Score 和交叉验证之后。我们就可以使用 sklearn 提供好了的接口来验证我们模型的性能了，代码十分简单。

```
# 导入K折功能
from sklearn.model_selection import KFold
from sklearn.metrics import f1_score

# 提取数据集中的标签
y = transactions['Class']
# 将Class这一列删除，也就相当于提取了数据集中的所有特征
X = transactions.drop(['Class'])

# 5折交叉验证
kf = KFold(n_splits=5)
# 平均f1 score
mean_f1_score = 0

# 开始5折交叉验证
for train_index, test_index in kf.split(X):
    #train_X表示训练集中的数据，train_y表示训练集中的标签
    train_X, train_y = X[train_index], y[train_index]
    #test_X表示验证集中的数据，test_y表示验证集中的标签
    test_X, test_y = X[test_index], y[test_index]
    smote_pipeline.fit(train_X, train_y)
    pred = smote_pipeline.predict(test_X)

    score = f1_score(test_y, pred)
    mean_f1_score += score

# 因为是5折，所以除以5
mean_f1_score /= 5
print(mean_f1_score)
```

```
0.8075666039570759
```

可以看到，我们的决策树模型的 F1 Score 为 0.8 左右。嗯，结果还是不错的。当然，我们还可以做更多的分析和处理工作，来让我们的分数越来越高。希望你能自己动手，尝试提高分数，相信你会享受这个过程的。

第十三章 综合实战：FIFA球员数据分析与推荐

在本书的最后，我们来做点更加有趣的事情。假设你现在是一支足球俱乐部的高层管理者，有一天，你的上司对你说：“我们俱乐部需要 2 名像梅西一样的球员来当替补”。那你接到这个任务后，我想你肯定会去分析球员的一些历史数据，然后来招募一些你所需要的球员。

我想你已经猜到了，没错！这次的数据是 FIFA 2019 年的数据。废话不多说，我们开始吧！

13.1 初步分析数据

这一步再熟悉不过了，可能会熟悉地让人心疼。但这又是数据挖掘非常重要也是必不可少的一步。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

datafr = pd.read_csv("./data.csv")

# 查看前10行数据到底长什么样子
datafr.head(10)
```

	Unnamed: 0	ID	Name	Age	Photo	Nationality	Flag
0	0	158023	L. Messi	31	https://cdn.sofifa.org/players/4/19/158023.png	Argentina	https://cdn.sofifa.org/flags/52.pr
1	1	20801	Cristiano Ronaldo	33	https://cdn.sofifa.org/players/4/19/20801.png	Portugal	https://cdn.sofifa.org/flags/38.pr
2	2	190871	Neymar Jr	26	https://cdn.sofifa.org/players/4/19/190871.png	Brazil	https://cdn.sofifa.org/flags/54.pr
3	3	193080	De Gea	27	https://cdn.sofifa.org/players/4/19/193080.png	Spain	https://cdn.sofifa.org/flags/45.pr
4	4	192985	K. De Bruyne	27	https://cdn.sofifa.org/players/4/19/192985.png	Belgium	https://cdn.sofifa.org/flags/7.png
5	5	183277	E. Hazard	27	https://cdn.sofifa.org/players/4/19/183277.png	Belgium	https://cdn.sofifa.org/flags/7.png
6	6	177003	L. Modrić	32	https://cdn.sofifa.org/players/4/19/177003.png	Croatia	https://cdn.sofifa.org/flags/10.pr
7	7	176580	L. Suárez	31	https://cdn.sofifa.org/players/4/19/176580.png	Uruguay	https://cdn.sofifa.org/flags/60.pr
8	8	155862	Sergio Ramos	32	https://cdn.sofifa.org/players/4/19/155862.png	Spain	https://cdn.sofifa.org/flags/45.pr
9	9	200389	J. Oblak	25	https://cdn.sofifa.org/players/4/19/200389.png	Slovenia	https://cdn.sofifa.org/flags/44.pr

Overall	Potential	Club	Club Logo	Value	Wage	Special	Preferred Foot	International Reputation
94	94	FC Barcelona	https://cdn.sofifa.org/teams/2/light/241.png	€110.5M	€565K	2202	Left	5.0
94	94	Juventus	https://cdn.sofifa.org/teams/2/light/45.png	€77M	€405K	2228	Right	5.0
92	93	Paris Saint-Germain	https://cdn.sofifa.org/teams/2/light/73.png	€118.5M	€290K	2143	Right	5.0
91	93	Manchester United	https://cdn.sofifa.org/teams/2/light/11.png	€72M	€260K	1471	Right	4.0
91	92	Manchester City	https://cdn.sofifa.org/teams/2/light/10.png	€102M	€355K	2281	Right	4.0
91	91	Chelsea	https://cdn.sofifa.org/teams/2/light/5.png	€93M	€340K	2142	Right	4.0
91	91	Real Madrid	https://cdn.sofifa.org/teams/2/light/243.png	€67M	€420K	2280	Right	4.0
91	91	FC Barcelona	https://cdn.sofifa.org/teams/2/light/241.png	€80M	€455K	2346	Right	5.0
91	91	Real Madrid	https://cdn.sofifa.org/teams/2/light/243.png	€51M	€380K	2201	Right	4.0
90	93	Atlético Madrid	https://cdn.sofifa.org/teams/2/light/240.png	€68M	€94K	1331	Right	3.0

Weak Foot	Skill Moves	Work Rate	Body Type	Real Face	Position	Jersey Number	Joined	Loaned From	Contract Valid Until	Height	Weight	LS	ST
4.0	4.0	Medium/ Medium	Messi	Yes	RF	10.0	Jul 1, 2004	NaN	2021	5'7	159lbs	88+2	88+2
4.0	5.0	High/ Low	C. Ronaldo	Yes	ST	7.0	Jul 10, 2018	NaN	2022	6'2	183lbs	91+3	91+3
5.0	5.0	High/ Medium	Neymar	Yes	LW	10.0	Aug 3, 2017	NaN	2022	5'9	150lbs	84+3	84+3
3.0	1.0	Medium/ Medium	Lean	Yes	GK	1.0	Jul 1, 2011	NaN	2020	6'4	168lbs	NaN	NaN
5.0	4.0	High/ High	Normal	Yes	RCM	7.0	Aug 30, 2015	NaN	2023	5'11	154lbs	82+3	82+3
4.0	4.0	High/ Medium	Normal	Yes	LF	10.0	Jul 1, 2012	NaN	2020	5'8	163lbs	83+3	83+3
4.0	4.0	High/ High	Lean	Yes	RCM	10.0	Aug 1, 2012	NaN	2020	5'8	146lbs	77+3	77+3
4.0	3.0	High/ Medium	Normal	Yes	RS	9.0	Jul 11, 2014	NaN	2021	6'0	190lbs	87+5	87+5
3.0	3.0	High/ Medium	Normal	Yes	RCB	15.0	Aug 1, 2005	NaN	2020	6'0	181lbs	73+3	73+3
3.0	1.0	Medium/ Medium	Normal	Yes	GK	1.0	Jul 16, 2014	NaN	2021	6'2	192lbs	NaN	NaN

RS	LW	LF	CF	RF	RW	LAM	CAM	RAM	LM	...	LB	LCB	CB	RCB	RB	Crossing
88+2	92+2	93+2	93+2	93+2	92+2	93+2	93+2	93+2	91+2	...	59+2	47+2	47+2	47+2	59+2	84.0
91+3	89+3	90+3	90+3	90+3	89+3	88+3	88+3	88+3	88+3	...	61+3	53+3	53+3	53+3	61+3	84.0
84+3	89+3	89+3	89+3	89+3	89+3	89+3	89+3	89+3	88+3	...	60+3	47+3	47+3	47+3	60+3	79.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	17.0
82+3	87+3	87+3	87+3	87+3	87+3	88+3	88+3	88+3	88+3	...	73+3	66+3	66+3	66+3	73+3	93.0
83+3	89+3	88+3	88+3	88+3	89+3	89+3	89+3	89+3	89+3	...	60+3	49+3	49+3	49+3	60+3	81.0
77+3	85+3	84+3	84+3	84+3	85+3	87+3	87+3	87+3	86+3	...	79+3	71+3	71+3	71+3	79+3	86.0
87+5	86+5	87+5	87+5	87+5	86+5	85+5	85+5	85+5	84+5	...	66+5	63+5	63+5	63+5	66+5	77.0
73+3	70+3	71+3	71+3	71+3	70+3	71+3	71+3	71+3	72+3	...	84+3	87+3	87+3	87+3	84+3	66.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	13.0

Finishing	HeadingAccuracy	ShortPassing	Volleys	Dribbling	Curve	FKAccuracy	LongPassing	BallControl	Acceleration
95.0	70.0	90.0	86.0	97.0	93.0	94.0	87.0	96.0	91.0
94.0	89.0	81.0	87.0	88.0	81.0	76.0	77.0	94.0	89.0
87.0	62.0	84.0	84.0	96.0	88.0	87.0	78.0	95.0	94.0
13.0	21.0	50.0	13.0	18.0	21.0	19.0	51.0	42.0	57.0
82.0	55.0	92.0	82.0	86.0	85.0	83.0	91.0	91.0	78.0
84.0	61.0	89.0	80.0	95.0	83.0	79.0	83.0	94.0	94.0
72.0	55.0	93.0	76.0	90.0	85.0	78.0	88.0	93.0	80.0
93.0	77.0	82.0	88.0	87.0	86.0	84.0	64.0	90.0	86.0
60.0	91.0	78.0	66.0	63.0	74.0	72.0	77.0	84.0	76.0
11.0	15.0	29.0	13.0	12.0	13.0	14.0	26.0	16.0	43.0

SprintSpeed	Agility	Reactions	Balance	ShotPower	Jumping	Stamina	Strength	LongShots	Aggression	Interceptions
86.0	91.0	95.0	95.0	85.0	68.0	72.0	59.0	94.0	48.0	22.0
91.0	87.0	96.0	70.0	95.0	95.0	88.0	79.0	93.0	63.0	29.0
90.0	96.0	94.0	84.0	80.0	61.0	81.0	49.0	82.0	56.0	36.0
58.0	60.0	90.0	43.0	31.0	67.0	43.0	64.0	12.0	38.0	30.0
76.0	79.0	91.0	77.0	91.0	63.0	90.0	75.0	91.0	76.0	61.0
88.0	95.0	90.0	94.0	82.0	56.0	83.0	66.0	80.0	54.0	41.0
72.0	93.0	90.0	94.0	79.0	68.0	89.0	58.0	82.0	62.0	83.0
75.0	82.0	92.0	83.0	86.0	69.0	90.0	83.0	85.0	87.0	41.0
75.0	78.0	85.0	66.0	79.0	93.0	84.0	83.0	59.0	88.0	90.0
60.0	67.0	86.0	49.0	22.0	76.0	41.0	78.0	12.0	34.0	19.0

Positioning	Vision	Penalties	Composure	Marking	StandingTackle	SlidingTackle	GKDiving	GKHandling	GKKicking
94.0	94.0	75.0	96.0	33.0	28.0	26.0	6.0	11.0	15.0
95.0	82.0	85.0	95.0	28.0	31.0	23.0	7.0	11.0	15.0
89.0	87.0	81.0	94.0	27.0	24.0	33.0	9.0	9.0	15.0
12.0	68.0	40.0	68.0	15.0	21.0	13.0	90.0	85.0	87.0
87.0	94.0	79.0	88.0	68.0	58.0	51.0	15.0	13.0	5.0
87.0	89.0	86.0	91.0	34.0	27.0	22.0	11.0	12.0	6.0
79.0	92.0	82.0	84.0	60.0	76.0	73.0	13.0	9.0	7.0
92.0	84.0	85.0	85.0	62.0	45.0	38.0	27.0	25.0	31.0
60.0	63.0	75.0	82.0	87.0	92.0	91.0	11.0	8.0	9.0
11.0	70.0	11.0	70.0	27.0	12.0	18.0	86.0	92.0	78.0

GKPositioning	GKReflexes	Release Clause
14.0	8.0	€226.5M
14.0	11.0	€127.1M
15.0	11.0	€228.1M
88.0	94.0	€138.6M
10.0	13.0	€196.4M
8.0	8.0	€172.1M
14.0	9.0	€137.4M
33.0	37.0	€164M
7.0	11.0	€104.6M
88.0	89.0	€144.5M

怎么样，是不是感觉有点懵，有这么多特征，还有一些缺失值。看来这个数据集并不是很好处理的样子。是的，我们需要一步一步脚印的来分析它。

还是按照惯例，看一下数据集有多少行多少列。

```
print("Dimension of the dataset is: ",datafr.shape)
```

```
Dimension of the dataset is:  (18207, 89)
```

总共 89 个特征！然后再看一下数据缺失的情况。

```
# 统计出含有缺失值的特征的数量
datafr.isnull().sum().sort_values(ascending=False)
```

Loaned From	16943
LWB	2085
LM	2085
CB	2085
LCB	2085
LB	2085
RWB	2085
RDM	2085
CDM	2085
LDM	2085
RM	2085
RCM	2085
CM	2085
LCM	2085
RAM	2085
RB	2085
CAM	2085
LAM	2085
RW	2085
RF	2085
CF	2085
LF	2085
LW	2085
RS	2085
ST	2085
LS	2085

```
RCB                       2085
Release Clause            1564
Joined                    1553
Contract Valid Until       289
                          ...
Positioning                48
FKAccuracy                 48
LongPassing                48
BallControl                48
Acceleration               48
SprintSpeed                48
Crossing                   48
Reactions                  48
ShotPower                  48
GKReflexes                 48
Jumping                    48
Stamina                    48
Strength                   48
LongShots                  48
Aggression                 48
Balance                    48
Interceptions              48
```

```
Flag                       0
ID                         0
Name                       0
Age                        0
Photo                      0
Nationality                0
Value                      0
Overall                    0
Potential                  0
Club Logo                  0
Wage                       0
Special                    0
Unnamed: 0                 0
Length: 89, dtype: int64
```

总共 89 个特征，也就只有 13 个特征是完好无损的。不过值得庆幸的是，含有缺失值的特征们的缺失比例并不高，只有 Loaned From 这个特征的缺失严重。所以我们暂且可以认为这个特征没有什么用处，把它删掉就好了。

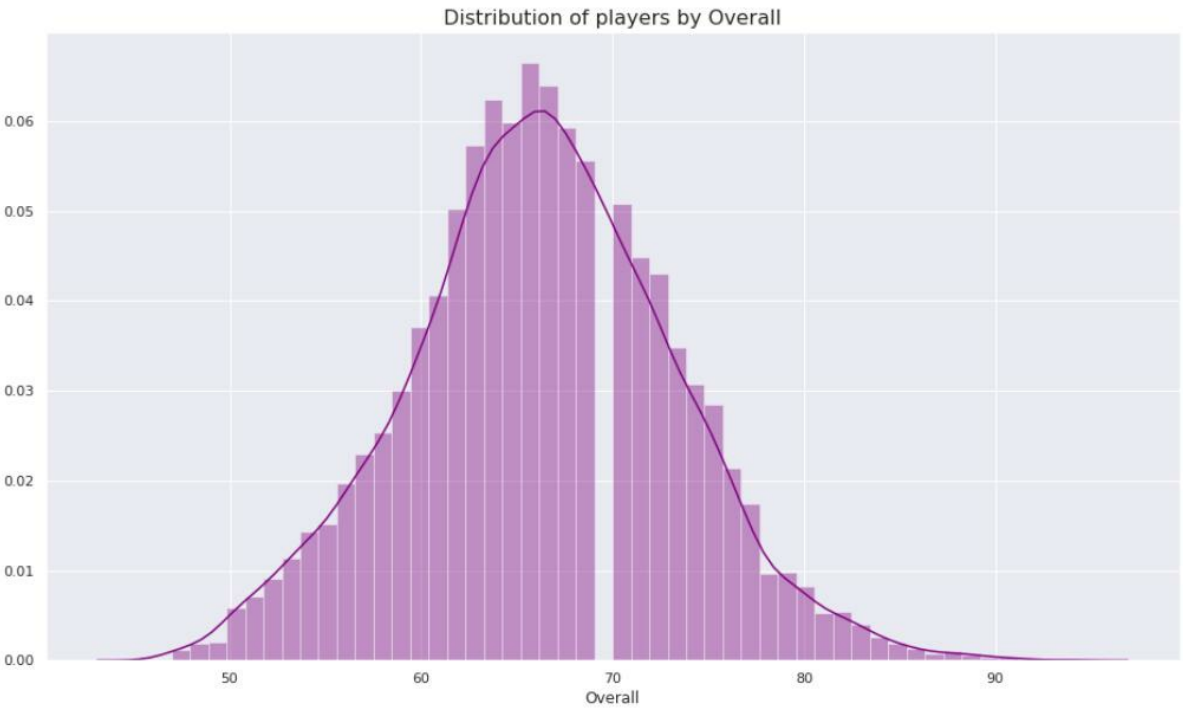
```
# 删除Loaned From
datafr.drop('Loaned From',axis=1,inplace=True)
```

11.2 FIFA数据可视化

可视化是实现理解数据的最好途径，所以接下来会对不同的特征进行可视化。

首先先看一下 Overall 这个特征，Overall 表示的是球员的综合水平，值越高说明球员越厉害。一般来说，球员的综合水平应该是符合高斯分布的。因为特别差劲与特别优秀的球员在所有球员中所占的比例应该是非常少的，水平中庸的蓝领球员应该是占大多数的。所以我们可以可视化看一下，球员的综合水平的整体分布是不是一个高斯分布。

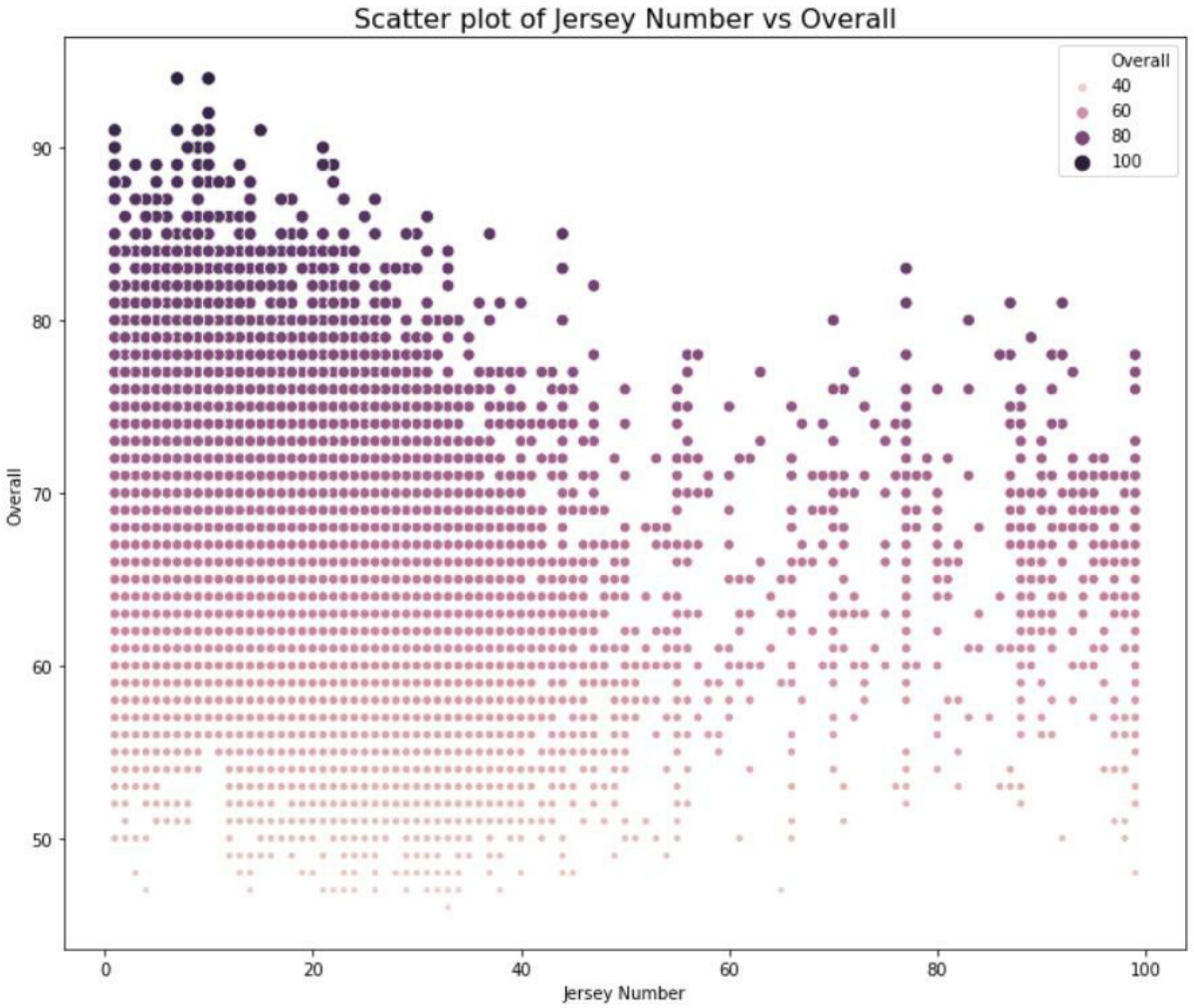
```
# 画分布图
sns.distplot(datafr['Overall'],color="Purple", ax=axs[1])
```



基本上符合高斯分布，所以基本可以推断出 Overall 这个特征应该没什么毛病。

然后来看一下 Jersey Number 这个特征，Jersey Number 的意思是球衣号码。为什么要看这个特征呢？因为我个人认为这种特征所代表的含义比较玄学，可能有些球衣号码的球员的普遍综合水平比较高。那为了验证这一想法，不如来可视化一下看看。

```
plt.figure(figsize=(12,10))

# 画散点图，横轴是球衣号码，纵轴是球员的综合水平
ax = sns.scatterplot(x="Jersey Number", y="Overall", hue ="Overall", size= "Overall", data=datafr)
ax.set_title('Scatter plot of Jersey Number vs Overall', fontsize=16)
sns.set_context("paper", font_scale=1.4)
```



从可视化的结果来看，球衣号码和球员的综合水平好想并没有比较直观的关系。所以如果想要根据球衣号码来推测出该球员的综合水平高低，不太现实。因此在后面就行推荐时可以将其删掉。

那么哪些特征会与球员的综合水平有关呢？我个人认为，球员的国籍可能是一个比较重要的特征。因为大部分绿茵场上的超级巨星都是来自如巴西、阿根廷、德国、意大利等国家。所以我们不妨来检查一下球员国籍这个特征有没有明显的异常值。

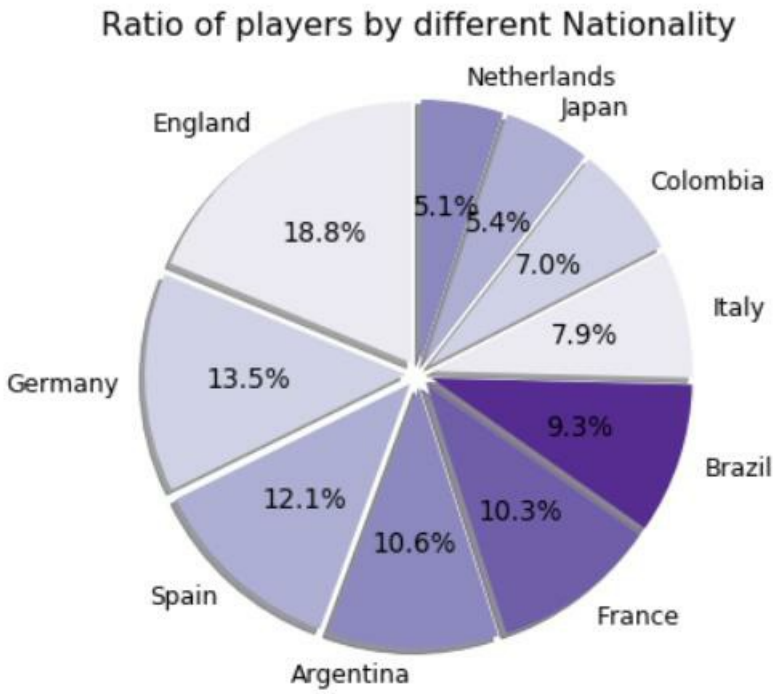
首先，看看出产球员最多的十个国家分别是哪些国家。

```
datafr['Nationality'].value_counts()[:10]
```

```
England        1662
Germany        1198
Spain          1072
Argentina       937
France          914
Brazil          827
Italy           702
Colombia        618
Japan           478
Netherlands     453
Name: Nationality, dtype: int64
```

和我预期的差不多，英格兰，德国，西班牙，法国，巴西，意大利等足球强国中出产的球员数量比较多，很符合常理，应该没有什么异常值。不过，干巴巴的数字看上去不够直观，我们可以将球员比例可视化出来。

```
England = len(datafr[datafr['Nationality'] == 'England'])
Germany = len(datafr[datafr['Nationality'] == 'Germany'])
Spain = len(datafr[datafr['Nationality'] == 'Spain'])
Argentina = len(datafr[datafr['Nationality'] == 'Argentina'])
France = len(datafr[datafr['Nationality'] == 'France'])
Brazil = len(datafr[datafr['Nationality'] == 'Brazil'])
Italy = len(datafr[datafr['Nationality'] == 'Italy'])
Colombia = len(datafr[datafr['Nationality'] == 'Colombia'])
Japan = len(datafr[datafr['Nationality'] == 'Japan'])
Netherlands = len(datafr[datafr['Nationality'] == 'Netherlands'])

labels = 'England','Germany','Spain','Argentina','France','Brazil','Italy','Colombia','Japan','Netherlands'
sizes = [England,Germany,Spain,Argentina,France,Brazil,Italy,Colombia,Japan,Netherlands]
plt.figure(figsize=(6,6))

# 画饼图
plt.pie(sizes, explode=(0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05), labels=labels, colors=sn
s.color_palette("Purples"),
autopct='%1.1f%%', shadow=True, startangle=90)
sns.set_context("paper", font_scale=1.2)
plt.title('Ratio of players by different Nationality', fontsize=16)
plt.show()
```



非常直观，这幅图其实也从侧面反映了各大足球强国的甲级联赛的风靡程度。

当然，还有很多特征与综合水平有关，在这里我就不一一列举了，就当是作业留给你自己去动手挖掘了。祝愿你能发现一些有趣的信息。

13.3 球员推荐

现在有这样一个需求，需要从球员库中找到与某位球员相似度比较高的几位球员。那这样一个需求应该怎样实现呢？我们可以回忆一下，本书中提到的算法中，哪个算法是与相似度有关的。没错！就是k近邻算法！所以我们只需要使用k近邻算法来找出离某位球员距离最近的 k 位球员就行了。不过在使用k近邻算法之前，我们需要对数据进行一些处理。

首先是删除一些无用的特征。

```
# 这些特征并没有太多用处，所以删掉。例如上一节所提到的球衣号码
datafr.drop(['ID','Unnamed: 0','Weak Foot','Release Clause','Wage','Photo', 'Nationality', 'Flag', 'Club Logo'
, 'International Reputation', 'Body Type', 'Real Face','Jersey Number', 'Joined','LS', 'ST', 'RS', 'LW', 'LF'
, 'CF', 'RF',  'RW','LAM', 'CAM', 'RAM', 'LM', 'LCM', 'CM', 'RCM', 'RM', 'LWB', 'LDM','CDM', 'RDM', 'RWB', 'L
B', 'LCB','CB', 'RCB', 'RB'], axis=1,inplace=True)

# 查看剩下的特征名称
datafr.columns
```

```
Index(['Name', 'Age', 'Overall', 'Potential', 'Club', 'Value', 'Special',
       'Preferred Foot', 'Skill Moves', 'Work Rate', 'Position',
       'Contract Valid Until', 'Height', 'Weight', 'Crossing', 'Finishing',
       'HeadingAccuracy', 'ShortPassing', 'Volleys', 'Dribbling', 'Curve',
       'FKAccuracy', 'LongPassing', 'BallControl', 'Acceleration',
       'SprintSpeed', 'Agility', 'Reactions', 'Balance', 'ShotPower',
       'Jumping', 'Stamina', 'Strength', 'LongShots', 'Aggression',
       'Interceptions', 'Positioning', 'Vision', 'Penalties', 'Composure',
       'Marking', 'StandingTackle', 'SlidingTackle', 'GKDiving', 'GKHandling',
       'GKKicking', 'GKPositioning', 'GKReflexes'],
      dtype='object')
```

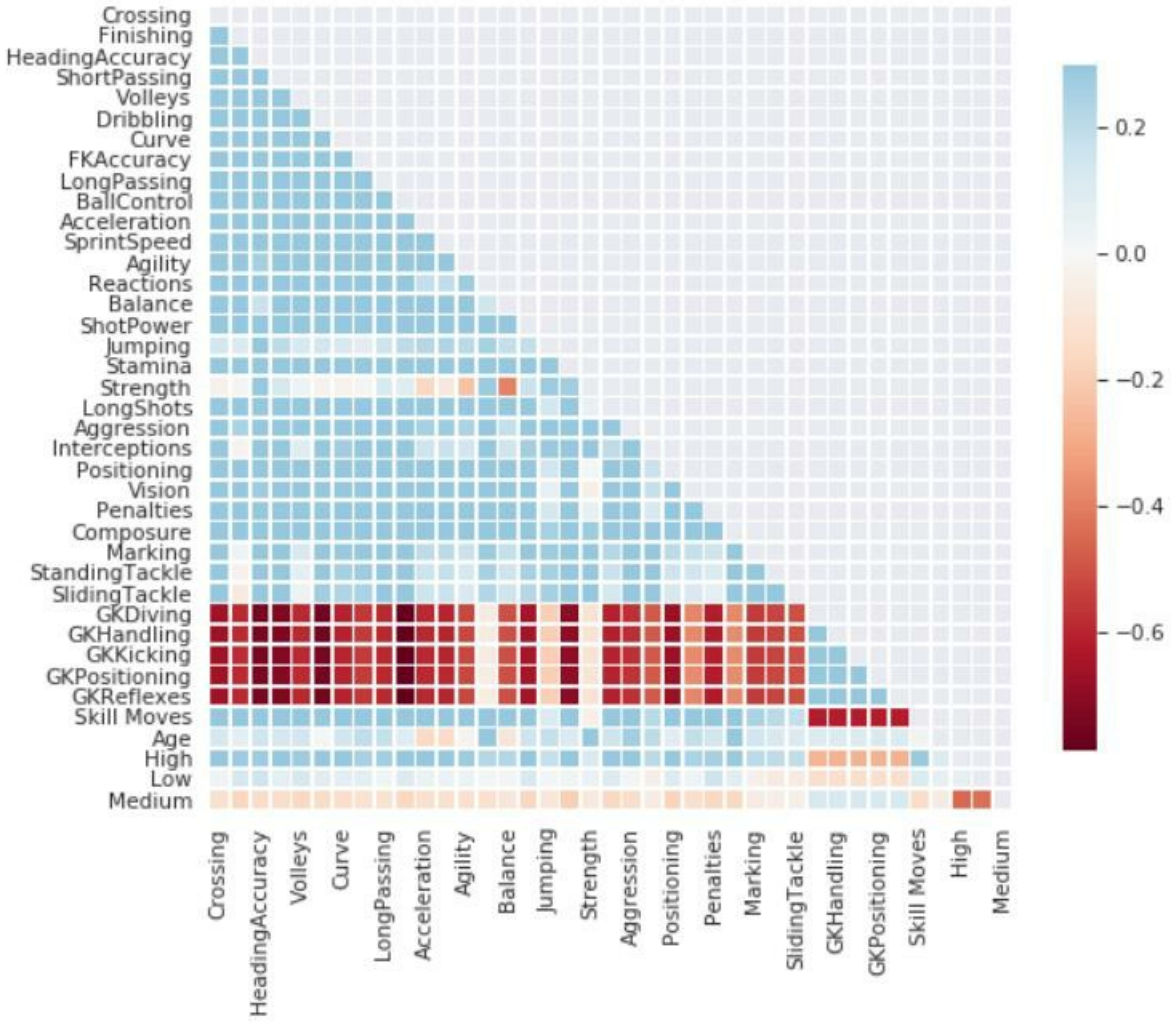
然后需要对剩下的特征做一些基本处理，比如离散化编码，处理缺失值等。

```
attributes = datafr.iloc[:, 14:]
attributes['Skill Moves'] = datafr['Skill Moves']
attributes['Age'] = datafr['Age']
# 独热编码
workrate = datafr['Work Rate'].str.get_dummies(sep='/ ')
attributes = pd.concat([attributes, workrate], axis=1)
df = attributes
# 删掉有缺失值的数据
attributes = attributes.dropna()
df['Name'] = datafr['Name']
df['Position'] = datafr['Position']
df = df.dropna()

print(attributes.columns)
```

```
Index(['Crossing', 'Finishing', 'HeadingAccuracy', 'ShortPassing', 'Volleys',
       'Dribbling', 'Curve', 'FKAccuracy', 'LongPassing', 'BallControl',
       'Acceleration', 'SprintSpeed', 'Agility', 'Reactions', 'Balance',
       'ShotPower', 'Jumping', 'Stamina', 'Strength', 'LongShots',
       'Aggression', 'Interceptions', 'Positioning', 'Vision', 'Penalties',
       'Composure', 'Marking', 'StandingTackle', 'SlidingTackle', 'GKDiving',
       'GKHandling', 'GKKicking', 'GKPositioning', 'GKReflexes', 'Skill Moves',
       'Age', 'High', 'Low', 'Medium'],
      dtype='object')
```

接着看一下这些特征的相关性。

```
plt.figure(figsize=(9,9))

# 计算特征之间的协方差
corr = attributes.corr()

# 因为相关性的热力图是对角线对称的，所以把上三角形遮掉看起来会清晰点
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# 画相关性热力图
sns.heatmap(corr, mask=mask, cmap="RdBu", vmax=.3, center=0,
            square=True, linewidths=.7, cbar_kws={"shrink": .7})
```



从上面的相关图可以看出，守门员的许多属性与前锋、中场和后卫的属性呈负相关。而且大部分特征之间的相关性比较小，算是比较独立的特征。

验证完相关性后，需要对特征进行标准化，来减少特征值的量纲所带来的影响，因为k近邻算法对量纲非常敏感。

```
# 导入sklearn提供的z-score标准化接口
from sklearn.preprocessing import StandardScaler

scaled = StandardScaler()
X = scaled.fit_transform(attributes)
```

最后，可以使用 sklearn 提供的k近邻算法接口来建模。

```
from sklearn.neighbors import NearestNeighbors

# 使用k近邻算法建模
recommendations = NearestNeighbors(n_neighbors=5,algorithm='kd_tree')
recommendations.fit(X)
```

有了模型之后，就可以实现相似球员的推荐了。

```
def get_index(x):
    return df[df['Name']==x].index.tolist()[0]

def recommend_similar(player):
    print("These are 4 players similar to {} : ".format(player))
    index=  get_index(player)
    for i in player_index[index][1:]:
        print("Name: {0}\nPosition: {1}\n".format(df.iloc[i]['Name'],df.iloc[i]['Position']))

# 查找与哈扎德相似的4名球员
recommend_similar('E. Hazard')
```

```
These are 4 players similar to E. Hazard :
Name: Neymar Jr
Position: LW

Name: P. Dybala
Position: LF

Name: Ronaldo Cabrais
Position: RW

Name: Douglas Costa
Position: LM
```

比利时的哈扎德，现在效力于皇家马德里。他的球风和能力与巴西的国脚内马尔是比较相近的。从结果可以看出，我们的推荐模型还是比较准确的。