## 数据归一化

### 作用

1.提高归一化速度

就是将数据除以该列的最大值，是该列<=1，数据归一化可以是等高线显得很圆，在梯度下降时能够较快收敛。（梯度下降时沿垂直等高线的方向，如果等高线很圆，就能很快指向圆心）

2.有可能提高精度

一些分类器需要计算样本间的距离，如果一个特征值范围非常大，那么计算距离就主要靠这个特征，可能与实际情况相悖。归一化可以有效解决这个问题。

3.试用范围

概率模型不需要归一化，因为他不关心变量的值，而是关心变量的分布和变量之间的条件概率，如决策树、rf。

而像adaboost、gbdt、xgboost、svm、lr、KNN、Kmeans之类的最优化问题需要归一化。

### 方法

#### 1.线性归一化

x’= 

#### 2.标准差归一化（0均值）





其中，μ、σ分别为原始数据集的均值和方法。该种归一化方式要求原始数据的分布可以近似为高斯分布，否则归一化的效果会变得很糟糕。

或者



#### 3.非线性归一化

应用于特征分布跨度很大，采用非线性归一化。通过一些函数，将原始值进行映射。一般采用log、指数函数、正切等函数。

#### 方法比较

1、在分类、聚类算法中，需要使用距离来度量相似性的时候、或者使用PCA技术进行降维的时候，第二种方法(Z-score standardization)表现更好。

2、在不涉及距离度量、协方差计算、数据不符合正太分布的时候，可以使用第一种方法或其他归一化方法。比如图像处理中，将RGB图像转换为灰度图像后将其值限定在[0 255]的范围。

## 梯度下降GD

梯度下降主要有三种

### 1.BGD

BGD 采用整个训练集的数据来计算 cost function 对参数的梯度。

**缺点：**

**由于这种方法是在一次更新中，就对整个数据集计算梯度，所以计算起来非常慢，遇到很大量的数据集也会非常棘手，而且不能投入新数据实时更新模型。**

### 2.SGD

和 BGD 的一次用所有数据计算梯度相比，SGD 每次更新时对每个样本进行梯度更新，对于很大的数据集来说，可能会有相似的样本，这样 BGD 在计算梯度时会出现冗余，而**SGD 一次只进行一次更新，就没有冗余，而且比较快，并且可以新增样本。**

随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况，那么可能只用其中部分的样本，就已经将theta迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代10次的话就需要遍历训练样本10次。

**缺点是SGD的噪音较BGD要多，使得SGD并不是每次迭代都向着整体最优化方向**。**所以虽然训练速度快，但是准确度下降，并不是全局最优**。**虽然包含一定的随机性，但是从期望上来看，它是等于正确的导数的。**

**缺点：**

**SGD 因为更新比较频繁，会造成 cost function 有严重的震荡。**

**BGD 可以收敛到局部极小值，当然 SGD 的震荡可能会跳到更好的局部极小值处。**

**当我们稍微减小 learning rate，SGD 和 BGD 的收敛性是一样的。**

### 3.MBGD

**MBGD 每一次利用一小批样本，即 n 个样本进行计算，这样它可以降低参数更新时的方差，收敛更稳定，另一方面可以充分地利用深度学习库中高度优化的矩阵操作来进行更有效的梯度计算。**

**超参数设定值:  n 一般取值在 50～256**

**缺点：（两大缺点）**

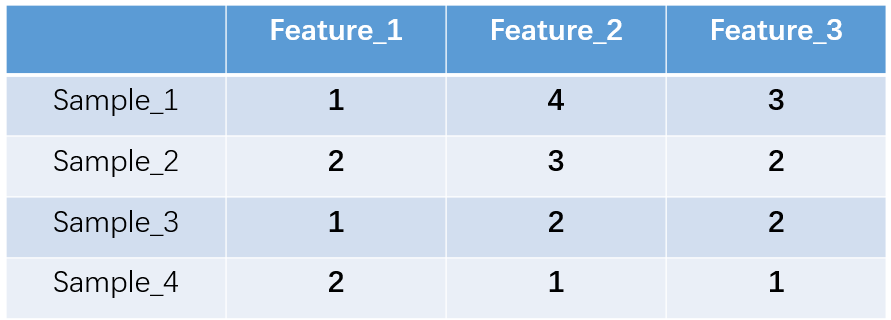
1. **不过 Mini-batch gradient descent 不能保证很好的收敛性，learning rate 如果选择的太小，收敛速度会很慢，如果太大，loss function 就会在极小值处不停地震荡甚至偏离。（有一种措施是先设定大一点的学习率，当两次迭代之间的变化低于某个阈值后，就减小 learning rate，不过这个阈值的设定需要提前写好，这样的话就不能够适应数据集的特点。）**对于非凸函数，还要避免陷于局部极小值处，或者鞍点处，因为鞍点周围的error是一样的，所有维度的梯度都接近于0，SGD 很容易被困在这里。（**会在鞍点或者局部最小点震荡跳动，因为在此点处，如果是训练集全集带入即BGD，则优化会停止不动，如果是mini-batch或者SGD，每次找到的梯度都是不同的，就会发生震荡，来回跳动。**）
2. SGD对所有参数更新时应用同样的 learning rate，如果我们的数据是稀疏的，**我们更希望对出现频率低的特征进行大一点的更新。LR会随着更新的次数逐渐变小。**

## 特征选取

### onehot

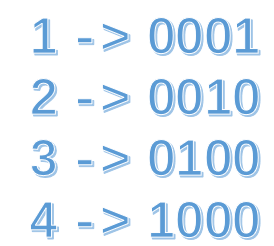
#### 1.onehot编码

什么是one-hot编码？one-hot编码，又称独热编码、一位有效编码。其方法是使用N位状态寄存器来对N个状态进行编码，每个状态都有它独立的寄存器位，并且在任意时候，其中只有一位有效。举个例子，假设我们有四个样本（行），每个样本有三个特征（列）。

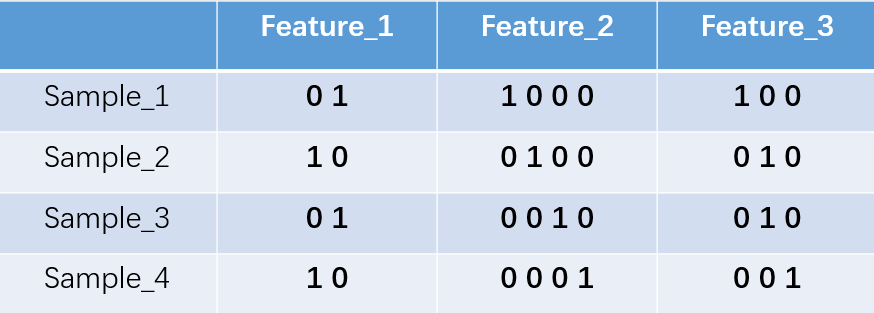


上图中我们已经对每个特征进行了普通的数字编码：我们的feature\_1有两种可能的取值，比如是男/女，这里男用1表示，女用2表示。那么one-hot编码是怎么搞的呢？我们再拿feature\_2来说明：

这里feature\_2 有4种取值（状态），我们就用4个状态位来表示这个特征，one-hot编码就是保证每个样本中的单个特征只有1位处于状态1，其他的都是0。



对于2种状态、三种状态、甚至更多状态都是这样表示，所以我们可以得到这些样本特征的新表示：



one-hot编码将每个状态位都看成一个特征。对于前两个样本我们可以得到它的特征向量分别为

#### C:\Users\Administrator\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.Word\1251096-20171030165731683-1946521226.png2 one-hot在提取文本特征上的应用

　　one hot在特征提取上属于词袋模型（bag of words）。关于如何使用one-hot抽取文本特征向量我们通过以下例子来说明。假设我们的语料库中有三段话：

　　　　我爱中国

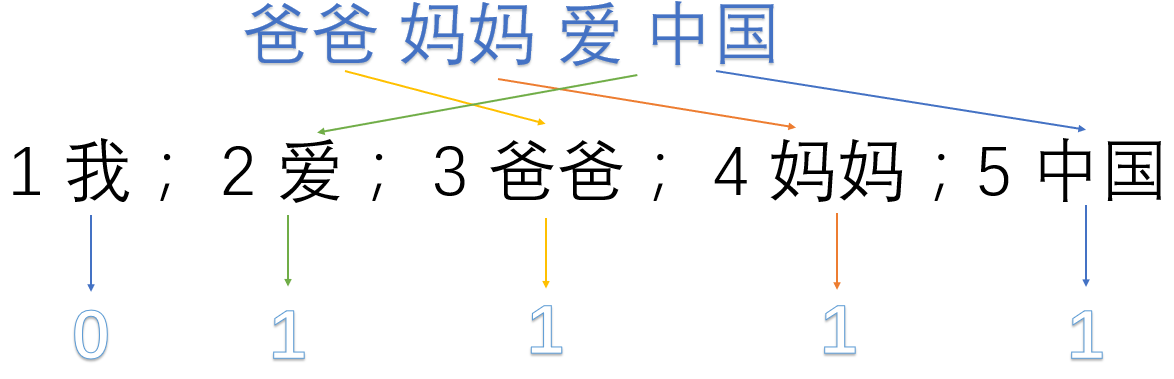
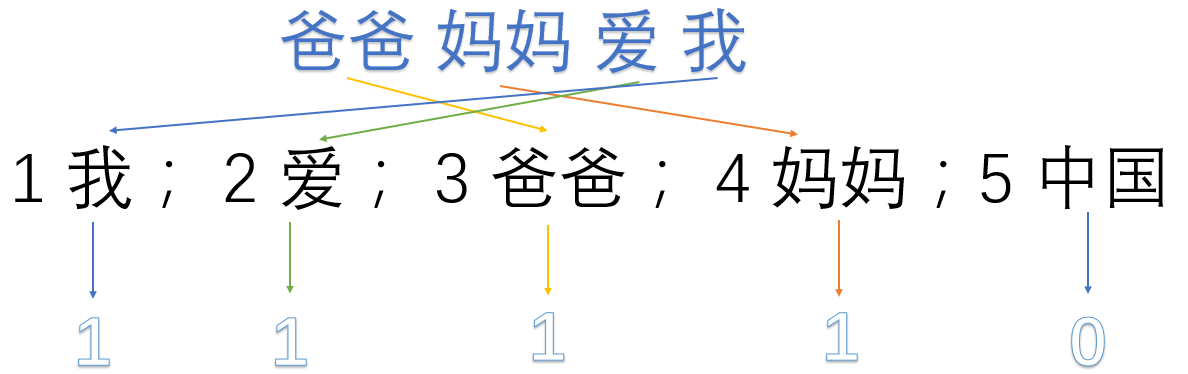
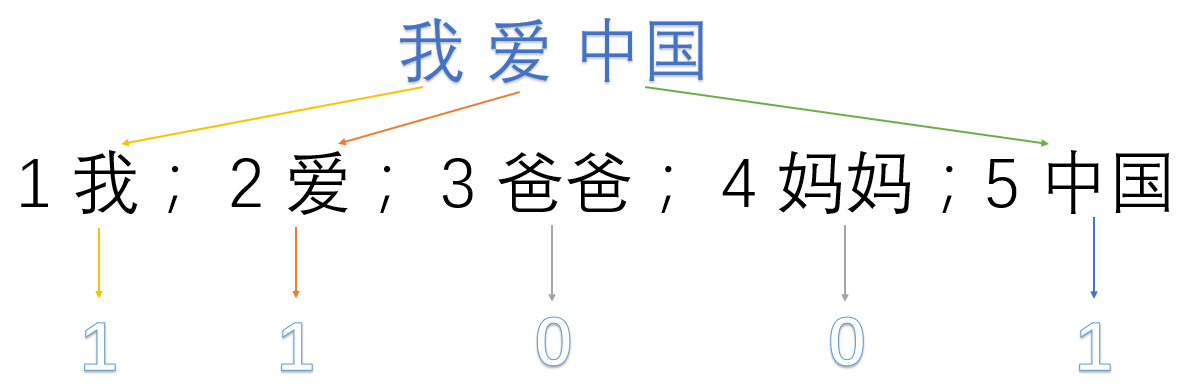
　　　　爸爸妈妈爱我

　　　　爸爸妈妈爱中国

我们首先对预料库分离并获取其中所有的词，然后对每个此进行编号：

　　　　1 我； 2 爱； 3 爸爸； 4 妈妈；5 中国

然后使用one hot对每段话提取特征向量：



因此我们得到了最终的特征向量为

　　　　我爱中国 　->　　　1，1，0，0，1

　　　　爸爸妈妈爱我　　->　　1，1，1，1，0

　　　　爸爸妈妈爱中国　　->　　0，1，1，1，1

优缺点分析

**优点：**一是解决了分类器不好处理离散数据的问题，二是在一定程度上也起到了扩充特征的作用（上面样本特征数从3扩展到了9）

**缺点：**在文本特征表示上有些缺点就非常突出了。首先，它是一个词袋模型，不考虑词与词之间的顺序（文本中词的顺序信息也是很重要的）；其次，它假设词与词相互独立（在大多数情况下，词与词是相互影响的）；最后，它得到的特征是离散稀疏的。

#### sklearn实现one hot encode

from sklearn import preprocessing

enc = preprocessing.OneHotEncoder() # 创建对象

enc.fit([[0,0,3],[1,1,0],[0,2,1],[1,0,2]]) # 拟合

array = enc.transform([[0,1,3]]).toarray() # 转化

print(array)

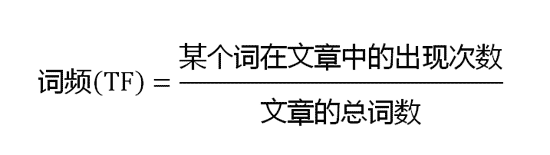
输入中，每一行为一个特征，每一列为一个输入。

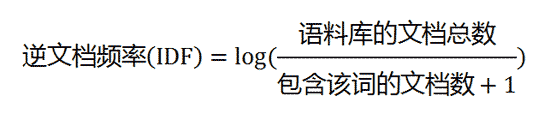
### 2. TF-IDF

#### 基本内容

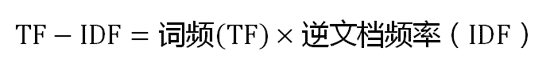
　　IF-IDF是信息检索（IR）中最常用的一种文本表示法。算法的思想也很简单，就是统计每个词出现的词频（TF），然后再为其附上一个权值参数（IDF）。举个例子：

　现在假设我们要统计一篇文档中的前10个关键词，应该怎么下手？首先想到的是统计一下文档中每个词出现的频率（TF），词频越高，这个词就越重要。但是统计完你可能会发现你得到的关键词基本都是“的”、“是”、“为”这样没有实际意义的词（停用词），这个问题怎么解决呢？你可能会想到为每个词都加一个权重，像这种”停用词“就加一个很小的权重（甚至是置为0），这个权重就是IDF。下面再来看看公式：





IF应该很容易理解就是计算词频，IDF衡量词的常见程度。为了计算IDF我们需要事先准备一个语料库用来模拟语言的使用环境，如果一个词越是常见，那么式子中分母就越大，逆文档频率就越小越接近于0。这里的分母+1是为了避免分母为0的情况出现。TF-IDF的计算公式如下：



根据公式很容易看出，TF-IDF的值与该词在文章中出现的频率成正比，与该词在整个语料库中出现的频率成反比，因此可以很好的实现提取文章中关键词的目的。

#### 优缺点分析

**优点：**简单快速，结果比较符合实际

**缺点：**单纯考虑词频，忽略了词与词的位置信息以及词与词之间的相互关系。

#### sklearn实现tfidf

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.feature\_extraction.text import TfidfTransformer

tag\_list = ['青年 吃货 唱歌',

'少年 游戏 叛逆',

'少年 吃货 足球']

vectorizer = CountVectorizer() #将文本中的词语转换为词频矩阵

X = vectorizer.fit\_transform(tag\_list) #计算个词语出现的次数

"""

word\_dict = vectorizer.vocabulary\_

{'唱歌': 2, '吃货': 1, '青年': 6, '足球': 5, '叛逆': 0, '少年': 3, '游戏': 4}

"""

transformer = TfidfTransformer()

tfidf = transformer.fit\_transform(X) #将词频矩阵X统计成TF-IDF值

print(tfidf.toarray())

<https://blog.csdn.net/m0_37324740/article/details/79411651>

将所有词都汇总在一个列表中，vectorizer.get\_feature\_names()

X返回的是一组元组，元组第一个元素表示的是输入文本所在的行，第二个元素表示该行的某个词在词汇总列表的位置。