# 机器学习03【机器学习】

# 主要内容

1. 数字聚类案例
2. TF-IDF算法
3. 微博营销案例
4. 逻辑回归算法
5. 音乐分类案例

# 学习目标

|  |  |  |
| --- | --- | --- |
| 节数 | 知识点 | 要求 |
| 第一节 （数字聚类案例） | K-Means数字聚类案例 | 掌握 |
| 使用matplotlib检验分类效果 | 掌握 |
| K-Means在SparkMllib中应用 | 掌握 |
| 第二节（TF-IDF算法） | TF-IDF算法 | 掌握 |
| 第三节（微博聚类案例） | 微博聚类案例分析 | 掌握 |
| 微博聚类代码实现 | 掌握 |
| 第四节（逻辑回归算法） | 逻辑回归算法 | 掌握 |
| 第五节（音乐分类案例） | 傅里叶变换 | 掌握 |
| 音乐分类案例 | 掌握 |

# 数字聚类案例

1. **K-means找中心点和数据点分类例子**

|  |
| --- |
| import numpy as np将每行数据放入一个数组内列表，返回一个二维列表  def **loadDataSet**(fileName): 建空列表  dataMat = []  fr = open(fileName)  for line in fr.readlines():  #按照制表符切割每行，返回一个列表list  curLine = line.strip().split(*'\t'*)切分后的每个列表中的元素，以float形式返回，map()内置函数，返回一个list  fltLine = map(float,curLine)  dataMat.append(fltLine)  return dataMat  离  def **distEclud**(vecA, vecB):  return np.sqrt(np.sum(np.power(vecA - vecB, 2)))  3个中心点的位置坐标，返回一个3\*2的矩阵  def **randCent**(dataSet, k):列数,2列  n = np.shape(dataSet)[1]  *'''*  *centroids是一个3\*2的矩阵，用于存储三个中心点的坐标*  *'''*  centroids = np.mat(np.zeros((k,n)))  for j in range(n):  #统计每一列的最小值  minJ = min(dataSet[:,j])  #每列最大值与最小值的差值  rangeJ = float(max(dataSet[:,j]) - minJ)  #random.rand(k,1) 产生k\*1的数组，里面的数据是0~1的浮点型。  array2 = minJ + rangeJ \* np.random.rand(k,1)  #转换成k\*1矩阵 赋值给centroids  centroids[:,j] = np.mat(array2)  return centroids  def **kMeans**(dataSet, k, distMeas=distEclud, createCent=randCent):  #计算矩阵所有行数 80  m = np.shape(dataSet)[0]  y.mat 将二维数组转换成矩阵  clusterAssment = np.mat(np.zeros((m,2)))  createCent找到K个随机中心点坐标  centroids = createCent(dataSet, k)  # print centroids  clusterChanged = True  while clusterChanged:  clusterChanged = False  #遍历80个数据到每个中心点的距离  for i in range(m):  #np.inf float的最大值，无穷大  minDist = np.inf  #当前点属于的类别号  minIndex = -1  #每个样本点到三个中心点的距离  for j in range(k):  返回两点距离的值  distJI = distMeas(centroids[j,:],dataSet[i,:])  if distJI < minDist:  #当前最小距离的值  minDist = distJI  #当前最小值属于哪个聚类  minIndex = j  #有与上次迭代计算的当前点的类别不相同的点  if clusterAssment[i,0] != minIndex:  clusterChanged = True  #将当前点的类别号和最小距离 赋值给clusterAssment的一行  clusterAssment[i,:] = minIndex,minDist  for cent in range(k):  ent[:,0].A==censInClust 取出的是对应是当前遍历cent类别的 所有行数据组成的一个矩阵  ptsInClust = dataSet[np.nonzero(clusterAssment[:,0].A==cent)[0]]心点坐标的位置  centroids[cent,:] = np.mean(ptsInClust, axis=0)  #返回 【 当前三个中心点的坐标】 【每个点的类别号，和到当前中心点的最小距离】  return centroids, clusterAssment  if \_\_name\_\_ == *'\_\_main\_\_'*: 80\*2的矩阵  dataMat = np.mat(loadDataSet(*'./testSet.txt'*))  k=3  centroids, clusterAssment =  kMeans(dataMat, k, distMeas=distEclud, createCent=randCent)  print centroids  print clusterAssment |

1. **使用matplotlib检验分类效果**

对于分类结果，可以抽样数据使用matplotlib检验数据的分布情况。

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from sklearn.cluster import KMeans  from sklearn.datasets import make\_blobs  #建立12\*12新的图像  plt.figure(figsize=(12, 12))  n\_samples = 1500  random\_state = 170  *'''*  *make\_blobs函数是为聚类产生数据集 ， 产生一个数据集和相应的标签*  *n\_samples:表示数据样本点个数,默认值100*  *n\_features:表示数据的维度，特征，默认值是2*  *centers:产生数据的中心点，默认值3个*  *shuffle ：洗乱，默认值是True*  *random\_state:官网解释是随机生成器的种子*  *'''*  #x返回的是向量化的数据点，y返回的是对应数据的类别号  x,y = make\_blobs(n\_samples=n\_samples, random\_state=random\_state)  #使用KMeans去聚类,返回聚好的类别集合,聚合成几类  y\_pred = KMeans(n\_clusters=3, random\_state=random\_state).fit\_predict(x)  #subplot 绘制多个子图，221 等价于2,2,1 表示两行两列的子图中的第一个  plt.subplot(221)  #scatter 绘制散点图  plt.scatter(x[:, 0], x[:, 1], c=y\_pred)  plt.title(*"kmeans01"*)    transformation = [[ 0.60834549, -0.63667341], [-0.40887718, 0.85253229]]  #numpy.dot 矩阵相乘  X\_aniso = np.dot(x, transformation)  y\_pred = KMeans(n\_clusters=3, random\_state=random\_state).fit\_predict(X\_aniso)  plt.subplot(222)  plt.scatter(X\_aniso[:, 0], X\_aniso[:, 1], c=y\_pred)  plt.title(*"kmeans02"*)    #vstack 是合并矩阵，将y=0类别的取出500行，y=1类别的取出100行，y=2类别的取出10行  X\_filtered = np.vstack((x[y == 0][:500], x[y == 1][:100], x[y == 2][:10]))  y\_pred = KMeans(n\_clusters=3, random\_state=random\_state).fit\_predict(X\_filtered)  plt.subplot(223)  plt.scatter(X\_filtered[:, 0], X\_filtered[:, 1], c=y\_pred)  plt.title(*"kmeans03"*)  dataMat = []  fr = open(*"testSet.txt"*,*"r"*)  for line in fr.readlines():  if line.strip() <> *""*:  curLine = line.strip().split(*'\t'*)  fltLine = map(float,curLine)  dataMat.append(fltLine)  dataMat = np.array(dataMat)  y\_pred = KMeans(n\_clusters=4, random\_state=random\_state).fit\_predict(dataMat)  plt.subplot(224)  plt.scatter(dataMat[:,0], dataMat[:, 1], c=y\_pred)  plt.title(*"kmeans04"*)  plt.savefig(*"./kmeans.png"*)  plt.show() |

1. **使用SparkMllib训练K-means模型**

|  |
| --- |
| **object** KMeans {  **def** main(args: Array[*String*]) {  //1 构建Spark对象  **val** conf = **new** SparkConf().setAppName("KMeans").setMaster("local")  **val** sc = **new** SparkContext(conf)  // 读取样本数据1，格式为LIBSVM format  **val** data = sc.textFile("kmeans\_data.txt")  **val** parsedData = data.map(s => Vectors.dense(s.split(' ')  .map(\_.toDouble))).cache()  **val** numClusters = 4  **val** numIterations = 100  **val** model = **new** KMeans().  //设置聚类的类数  setK(numClusters).  //设置找中心点最大的迭代次数  setMaxIterations(numIterations).  run(parsedData)    //四个中心点的坐标  **val** centers = model.clusterCenters  **val** k = model.k  centers.foreach(println)  println(k)  //保存模型  // model.save(sc, "./Kmeans\_model")  //加载模型  **val** sameModel = KMeansModel.load(sc, "./Kmeans\_model")  println(sameModel.predict(Vectors.dense(1,1,1)))  **val** sqlContext = **new** SQLContext(sc)  sqlContext.read.parquet("./Kmeans\_model/data").show()    }  } |

给Kmeans指定中心点坐标：

|  |
| --- |
| **object** KMeans2 {  **def** main(args: Array[*String*]) {  **val** conf = **new** SparkConf().setAppName("KMeans2").setMaster("local")  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.parallelize(List(  Vectors.dense(Array(-0.1, 0.0, 0.0)),  Vectors.dense(Array(9.0, 9.0, 9.0)),  Vectors.dense(Array(3.0, 2.0, 1.0))))    //指定文件 kmeans\_data.txt 中的六个点为中心点坐标。  **val** centroids: Array[Vector] = sc.textFile("kmeans\_data.txt")  .map(\_.split(" ").map(\_.toDouble))  .map(Vectors.dense(\_))  .collect()  **val** model = **new** KMeansModel(clusterCenters=centroids)  println("聚类个数 = "+model.k)  //模型中心点  model.clusterCenters.foreach { println }  //预测指定的三条数据  **val** result = model.predict(rdd)  result.collect().foreach(println(\_))  }  } |

# TF-IDF算法

1. **TF-IDF**

TF-IDF(Term Frequency-Inverse Document Frequency, 词频-逆文本频率)。TF指词频，IDF指的是逆文本频率。TF-IDF是一种用于信息检索与数据挖掘的常用加权技术，可以评估一个词在一个文件集或者一个语料库中对某个文件的重要程度。一个词语在一篇文章中出现的次数越多，同时在所有文章中出现的次数越少，越能够代表该文章的中心意思，这就是TF-IDF的含义。

词频（Term Frequency ,TF）指的是一个给定的词语在该文件中出现的次数。这个数字通常会被归一化（一般是词频除以文章总词数），来防止这个词偏向内容较长文件。比如：同一个词语在长文件里可能会比短文件有更高的词频。



逆向文本频率（Inverse Document Frequency，IDF）,主要思想是在语料库中包含某个词条的文档越少，该词条IDF越大，说明这个词就有很强的类别区分能力。通常某个词语的IDF可以由语料库中文件的总数量除以包含该词语的文件数目，再将得到的商取对数决定。



分母加1主要是避免这个单词在语料库文章中没有出现，避免分母为0。这样当一个词语在语料库少数文章中出现，该词语对应的IDF值越大，当一个词语在大多数文章中都出现，那么该词语对应的IDF值会越小。

这样，某个词条的TF-IDF的值为 ，即某一个文件中高频出现的词条，以及该词条在整个语料库文件中低频出现的现象，就可以产生高权重的TF-IDF,因此，TF-IDF倾向过滤掉常见的词语，保留重要的词语。即：字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。

# 微博聚类案例

1. **微博聚类分析**

要实现广告的精准投放，需要使用聚类找出兴趣一致的用户群体，这样就需要对用户进行聚类找出行为一致的用户，当对所有用户完成聚类之后，再使用关键词分析找出每个聚类群体中的用户的讨论主题，如果主题符合广告内容或者和广告内容相关，那么当前广告就可以推荐给当前用户群体，实现精准投放广告。

1. **微博营销案例**

|  |
| --- |
| **object** KMeans11 {  **def** main(args: Array[*String*]) {  **val** conf = **new** SparkConf().setAppName("KMeans1").setMaster("local[\*]")  **val** sc = **new** SparkContext(conf)  **val** rdd = sc.textFile("./original.txt")  /\*\*  \* wordRDD 是一个KV格式的RDD  \* K:微博ID  \* V:微博内容分词后的结果 ArrayBuffer  \*/  **var** wordRDD = rdd.mapPartitions(iterator => {  **val** list = **new** ListBuffer[(*String*, ArrayBuffer[*String*])]  **while** (iterator.hasNext) {  //创建分词对象  **val** analyzer = **new** IKAnalyzer(**true**)  **val** line = iterator.next()  **val** textArr = line.split("\t")  **val** id = textArr(0)  **val** text = textArr(1)  //分词  **val** ts : TokenStream = analyzer.tokenStream("", text)  **val** term : CharTermAttribute = ts.getAttribute(classOf[CharTermAttribute])  ts.reset()    **val** arr = **new** ArrayBuffer[*String*]  **while** (ts.incrementToken()) {  arr.+=(term.toString())  }    list.append((id, arr))  analyzer.close()  }  list.iterator  })  wordRDD = wordRDD.cache()    /\*\*  \* HashingTF 使用hash表来存储分词  \*  \* 1000:只是计算每篇微博中1000个单词的词频 最大似然估计思想  \*/  **val** hashingTF: HashingTF = **new** HashingTF(1000)  /\*\*  \* tfRDD  \* K:微博ID  \* V:Vector（tf，tf，tf.....）  \*  \* hashingTF.transform(x.\_2) 计算分词频数（TF）  \*/  **val** tfRDD = wordRDD.map(x => {  (x.\_1, hashingTF.transform(x.\_2))  })  /\*\*  \* 得到IDFModel，要计算每个单词在整个语料库中的IDF  \*/  **val** idf: IDFModel = **new** IDF().fit(tfRDD.map(\_.\_2))  /\*\*  \* K:weibo ID  \* V:每一个单词的TF-IDF值  \* tfIdfs这个RDD中的Vector就是训练模型的训练集  \*  \*/  **val** tfIdfs: RDD[(*String*, Vector)] = tfRDD.mapValues(idf.transform(\_))    //设置聚类个数  **val** kcluster = 20  **val** kmeans = **new** KMeans()  kmeans.setK(kcluster)  //使用的是kemans++算法来训练模型  kmeans.setInitializationMode("k-means||")  //设置最大迭代次数  kmeans.setMaxIterations(100)  **val** kmeansModel: KMeansModel= kmeans.run(tfIdfs.map(\_.\_2))  // kmeansModel.save(sc, "d:/model001")  //打印模型的20个中心点  println(kmeansModel.clusterCenters)  /\*\*  \* 模型预测  \*/  **val** modelBroadcast = sc.broadcast(kmeansModel)  /\*\*  \* predicetionRDD KV格式的RDD  \* K：微博ID  \* V：分类号  \*/  **val** predicetionRDD = tfIdfs.mapValues(sample => {  **val** model = modelBroadcast.value  model.predict(sample)  })  // predicetionRDD.saveAsTextFile("d:/resultttt")  /\*\*  \* 总结预测结果  \* tfIdfs2wordsRDD:kv格式的RDD  \* K：微博ID  \* V：二元组(Vector(tfidf1,tfidf2....),ArrayBuffer(word,word,word....))  \*/  **val** tfIdfs2wordsRDD = tfIdfs.join(wordRDD)  /\*\*  \* result:KV  \* K：微博ID  \* V:(类别号，(Vector(tfidf1,tfidf2....),ArrayBuffer(word,word,word....)))  \*/  **val** result = predicetionRDD.join(tfIdfs2wordsRDD)  /\*\*  \* 查看0号类别中tf-idf比较高的单词，能代表这类的主题  \*/  result  .filter(x => x.\_2.\_1 == 0)  .flatMap(line => {    **val** tfIdfV: Vector = line.\_2.\_2.\_1  **val** words: ArrayBuffer[*String*] = line.\_2.\_2.\_2  **val** tfIdfA: Array[Double] = tfIdfV.toArray    **val** wordL = **new** ListBuffer[*String*]()  **val** tfIdfL = **new** ListBuffer[Double]()  **var** index = 0  **for**(i <- 0 until tfIdfA.length ;**if** tfIdfV(i) != 0){  wordL.+=(words(index))  tfIdfL.+=(tfIdfA(index))  index += 1  }  println(wordL.length + "===" + tfIdfL.length)  **val** list = **new** ListBuffer[(Double, *String*)]  **for** (i <- 0 until wordL.length) {  list.append((tfIdfV(i), words(i)))  }  list  })  .sortBy(x => x.\_1, **false**)  .map(\_.\_2)  .distinct()  .take(30).foreach(println)  sc.stop()  }  } |

# 逻辑回归算法

1. **逻辑回归**

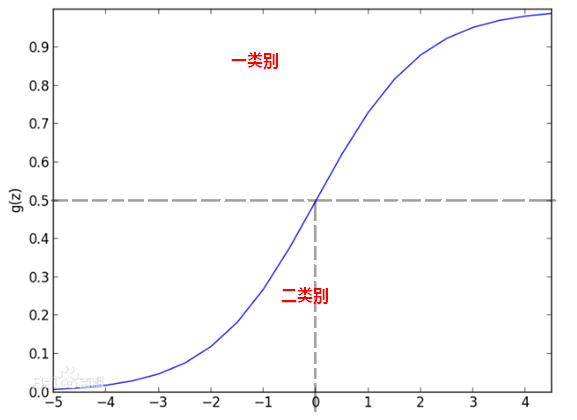
逻辑回归又叫logistic回归分析，是一种广义的线性回归分析模型。线性回归要求因变量必须是连续性的数据变量，逻辑回归要求因变量必须是分类变量，可以是二分类或者多分类(多分类都可以归结到二分类问题)，逻辑回归的输出是0~1之间的概率。比如要分析年龄，性别，身高，饮食习惯对于体重的影响，如果体重是实际的重量，那么就要使用线性回归。如果将体重分类，分成了高，中，低三类，就要使用逻辑回归进行分类。

逻辑回归公式：

其中，e是自然对数,无限不循环小数，

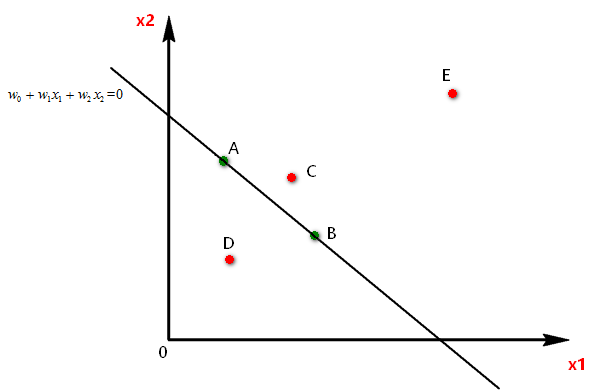


逻辑回归公式又叫逻辑函数（Logistic function）或者S形函数（Sigmoid function）。逻辑回归公式的图像如下：



即当z=0时， =0.5，当z 时，趋近于1，当z时，趋近于0。逻辑回归的输出就是位于0~1之间的概率，假设现在判断病人是否生病，得到的z=2对应的=0.7我们可以归结为生病，如果z=-2对应的=0.1我们就可以认为不生病。当z=0时，=0.5是决策的边界。

假设，，，都大于零，那么当z=0时，=0.5。也就是时，=0.5，当z=0时使用图像来表达两个自变量的关系为：



图中A、B、C、D、E点都表示有，两个维度的数据，现要使用逻辑回归对五个点分成两类：I和II类，A点和B点都在直线上，对应的z值是0，那么逻辑回归结果=0.5，将A,B两点划分为I类和II类都可以，假设我们规定当z>=0属于I类，z<0属于II类，那么A,B属于I类。C点位于直线的上方，对应的z值要大于零，反映到S形函数上对应的>0.5属于I类，同理，E也属于I类，D点属于II类。图中E点的z值远大于C点的z值，反映到S形函数中，E点属于I类的概率比C点属于I类的概率要大的多。训练逻辑回归模型，在这里就是训练出一条直线将两个类别的点隔开。如果维度是3,那么训练逻辑回归模型就是训练一个平面将两个类别的点隔开。如果维度大于3,那么训练逻辑回归模型就是训练一个超平面将两个类别的点隔开。

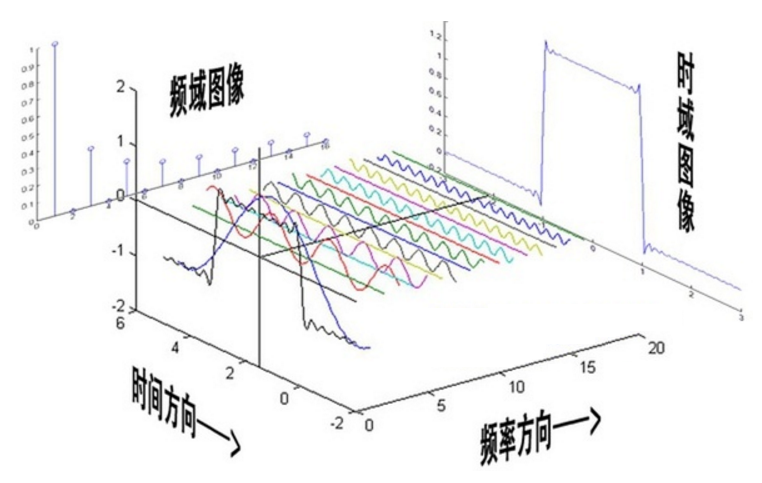
# 音乐分类案例

1. **傅里叶变换：**

时域分析：对一个信号来说，信号强度随时间的变化的规律就是时域特性，例如一个信号的时域波形可以表达信号随着时间的变化。

频域分析：对一个信号来说，在对其进行分析时，分析信号和频率有关的部分，而不是和时间相关的部分，和时域相对。也就是信号是由哪些单一频率的的信号合成的就是频域特性。频域中有一个重要的规则是正弦波是频域中唯一存在的波。即正弦波是对频域的描述，因为时域中的任何波形都可用正弦波合成。

一般来说，时域的表示较为形象直观，频域分析则简练。傅里叶变换是贯穿时域和频域的方法之一，傅里叶变换就是将难以处理的时域信号转换成了易于分析的频域信号。傅里叶原理：任何连续测量的时序信号，都可以表示为不同频率的正弦波信号的无限叠加。



1. **音乐分类的步骤:**
2. 通过傅里叶变换将不同7类里面所有原始wav格式音乐文件转换为特征,并取前1000个特征,存入文件以便后续训练使用
3. 读入以上7类特征向量数据作为训练集
4. 使用sklearn包中LogisticRegression的fit方法计算出分类模型
5. 读入黑豹乐队歌曲”无地自容”并进行傅里叶变换同样取前1000维作为特征向量
6. 调用模型的predict方法对音乐进行分类,结果分为rock即摇滚类
7. **代码：**

|  |
| --- |
| from scipy import fft  from scipy.io import wavfile  from scipy.stats import norm  from sklearn import linear\_model, datasets  from sklearn.linear\_model import LogisticRegression  import matplotlib.pyplot as plt  import numpy as np  *"""*  *使用logistic regression处理音乐数据,音乐数据训练样本的获得和使用快速傅里叶变换（FFT）预处理的方法需要事先准备好*  *1. 把训练集扩大到每类100个首歌,类别仍然是六类:jazz,classical,country, pop, rock, metal*  *2. 同时使用logistic回归训练模型*  *3. 引入一些评价的标准来比较Logistic测试集上的表现*  *"""*  # 准备音乐数据  def **create\_fft**(g,n):  rad=*"i:/genres/"*+g+*"/converted/"*+g+*"."*+str(n).zfill(5)+*".au.wav"*  (sample\_rate, X) = wavfile.read(rad)  #取1000个频率特征  fft\_features = abs(fft(X)[:1000])  #zfill(5) 字符串不足5位，前面补0  sad=*"i:/trainset/"*+g+*"."*+str(n).zfill(5)+ *".fft"*  np.save(sad, fft\_features)  genre\_list = [*"classical"*, *"jazz"*, *"country"*, *"pop"*, *"rock"*, *"metal"*,*"hiphop"*]  for g in genre\_list:  for n in range(100):  create\_fft(g,n)  print *'running...'*  print *'finished'*  read fft--------------  genre\_list = [*"classical"*, *"jazz"*, *"country"*, *"pop"*, *"rock"*, *"metal"*,*"hiphop"*]  X=[]  Y=[]  for g in genre\_list:  for n in range(100):  rad=*"i:/trainset/"*+g+*"."*+str(n).zfill(5)+ *".fft"*+*".npy"*  fft\_features = np.load(rad)  X.append(fft\_features)  #genre\_list.index(g) 返回匹配上类别的索引号  Y.append(genre\_list.index(g))    #构建的训练集  X=np.array(X)  #构建的训练集对应的类别  Y=np.array(Y)  -----train logistic classifier--------------  model = LogisticRegression()  #需要numpy.array类型参数  model.fit(X, Y)    print *'Starting read wavfile...'*  #prepare test data-------------------  sample\_rate, test = wavfile.read(*"i:/classical.00007.au.wav"*)  # sample\_rate, test = wavfile.read("i:/heibao-wudizirong-remix.wav")  testdata\_fft\_features = abs(fft(test))[:1000]  type\_index = model.predict(testdata\_fft\_features)[0]  print type\_index  print genre\_list[type\_index] |

# 本节作业

1. 掌握TF-IDF算法。
2. 掌握逻辑回归算法原理。