# MLLIB库特征选择文档

## TF-IDF

### TF-IDF介绍

TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。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。TF-IDF加权的各种形式常被搜索引擎应用，作为文件与用户查询之间相关程度的度量或评级。

1. TF(t,d)的含义是一个词term在某个文档document中出现的次数；
2. IDF计算公式: 

其中|D|代表所有文档的个数；DF(t,D)是包含该term的文档的个数。

1. TF-IDF(t,d,D)= TF(t,d) \* IDF(t,D)

### TF-IDF使用

1. 首先可以通过读取文件数据或者已有的RDD作为待处理的数据集documents，需要形成的数据集格式是RDD[Iterable的实现类] documents；
2. 其次，对这个RDD进行处理，通过org.apache.spark.mllib.feature.HashingTF类将其转换为RDD[Vector]形式的RDD，代码如下：

HashingTF hashingTF = new HashingTF()

JavaRDD<Vector> tf = hashingTF.transform(documents)

通过这两行代码我们可以得到TF(t,d)，这个RDD每一行代表一个文档，向量里面每一个值代表该term在该文档出现的次数；

1. 然后通过下面代码可以得到IDF(t,D)，

IDFModel idf = new IDF().fit(tf);

1. 最后通过下面代码我们可以得到每个term在每个文档中的重要度

JavaRDD<Vector> transform = idf.transform(tf);

### 示例

数据集RDD[List<String>] 如下包含了两个文档一行代表一个文档：

I am a student He is a doctor

I love my home and my family

针对上面的数据集HashingTf会将RDD中的每一行计算该行每个词才该文档出现的次数结果会输出一个RDD[Vector]向量，稀疏向量类型，每个单词已经被索引化。比如第一个文档I am a student He is a doctor将会得到

Index(I)->1 Index(am)->1 Index(student)->1 Index(He)->1 Index(is)->1 Index(a)->2 Index(doctor)->1

这一行被存放到一个稀疏向量里面，包含了一个索引数组一个数值数组

第二个文档同理最后生成RDD[Vector]格式。

最终经过IDF处理得到IDFModel里面包含了一个Vector其中也是包含了每个词的索引及其改词出现在文档中的文档数目，比如单词I 记录如下Index(I)->2 比如单词my记录如下Index(my)->1

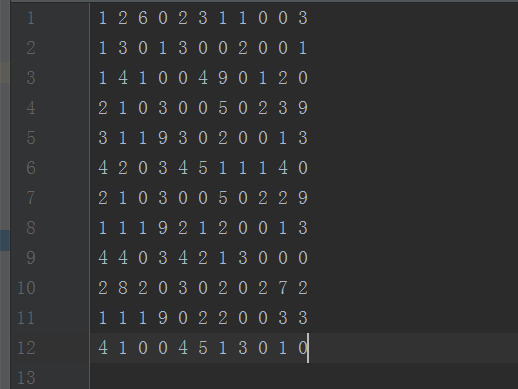
之后我们计算这里面|D|=2 2个文档

比如单词my IDF(my,D)=log((2+1)/(1+1))=log(3/2)

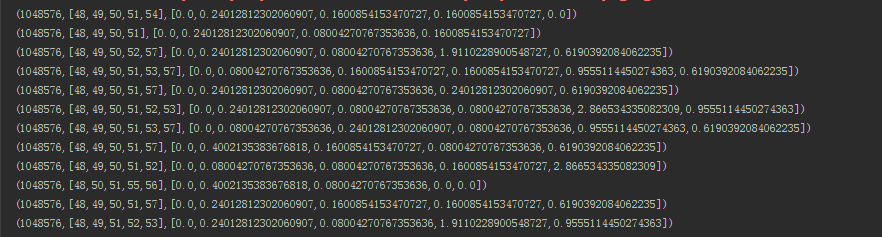
然后我们计算my的TFIDF我们需要用每个文档里面my的出现次数\*IDF(my,D)

因为第一个文档中没有my因此不需要计算，第二个文档中有my我们计算它的TFIDF(my,D)=TF(my,2)\*IDF(my,D)=2\* log(3/2)

最终我们可以得到每个文本中每个词的TFIDF值，结果形式也是RDD<Vector>

比如数据集如下：

输出TFIDF结果如下：第一列是文档的标号，然后接着一个数组是存放的每一行单词的索引 下面的数组存放的是与之对应的TFIDF的值



## Word2Vec

### Word2Vec介绍

Word2Vec是用来计算单词的向量表示的方法，相似单词的表示在向量空间是更加相似的，我们通过训练的模型寻找与某个单词最为接近的k个单词。

### 2.2 Word2Vec使用

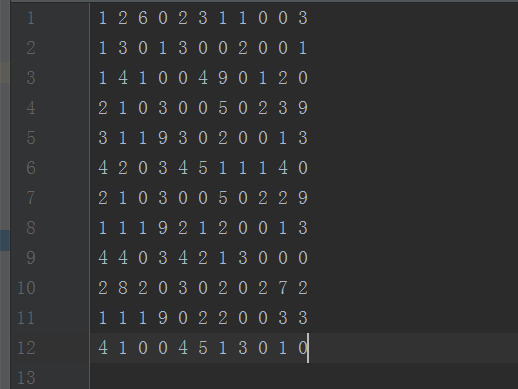
1. 数据集格式依旧是RDD[Iterable的实现类]类型的RDD数据集变量叫做documents
2. Word2Vec word2vec=new Word2Vec();

Word2VecModel model = word2vec.fit(documents);通过训练可以得到word2vec的model

1. model.getVectors我们可以得到每个单词的向量表示
2. model. findSynonyms(Stirng s,int k)我们可以得到与单词s相近的k个单词
3. model.transform(String word),返回一个向量 获取这个单词的向量表达，默认的vectorSize=100就是得到一个100维度表示该单词的向量

### 示例

比如输入数据集如下：



Word2Vec word2vec=new Word2Vec();  
Word2VecModel model = word2vec.fit(documents);  
  
 Tuple2<String, Object>[] synonyms = model.findSynonyms("1", 2);  
 for (Tuple2 t:  
 synonyms) {  
 System.*out*.println(t.\_1()+" "+t.\_2());  
 }

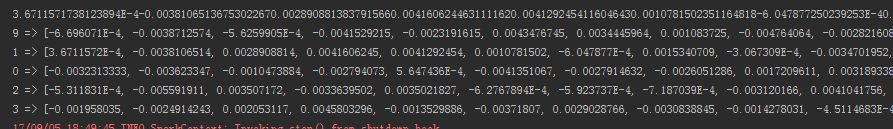
通过上面代码我们去寻找与单词“1”最为接近的两个单词，得到结果如下：



如果获取所有单词的向量表达使用如下代码：

Map<String, float[]> vectors1 = model.getVectors();  
 java.util.Map<String, float[]> stringMap1 = JavaConverters.*asJavaMapConverter*(vectors1).asJava();  
  
  
for(java.util.Map.Entry<String, float[]> entry : stringMap1.entrySet()){  
 System.*out*.println(String.*format*("%s => %s", entry.getKey(), Arrays.*toString*(entry.getValue())));  
}

结果如下图：



## 3 StandardScaler特征标准化

### 3.1 StandardScaler的介绍

这是一个预处理数据的步骤，一般通过使用缩放到单位方差或者减去均值 或者两者一起使用的方法来标准化特征。

### 3.2 StandardScaler的使用

默认情况下withMean为false withStd为true

**注意事项：**

数据格式是RDD<Vector>格式才可以调用fit的方法，如果我们是JavaRDD需要转为RDD才可以；

其次我们调用StandardScalerModel的transform时候可以是JavaRDD<Vector>类型

**参数解释**：

1: 使用withStd默认为true

scaler1 = new StandardScaler().fit(data.map(x => x.features))

2：使用withStd和withMean

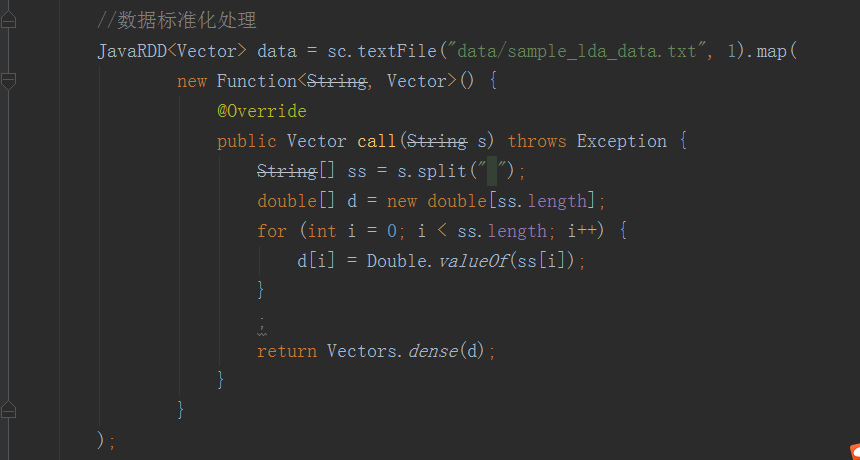
scaler2 = new StandardScaler(withMean = true, withStd = true).fit(data.map(x => x.features))

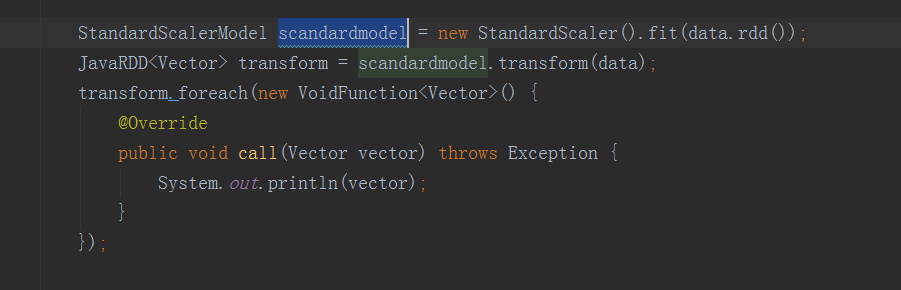
3: //scaler3与scaler2转换一样

val scaler3 = new StandardScalerModel(scaler2.std, scaler2.mean)

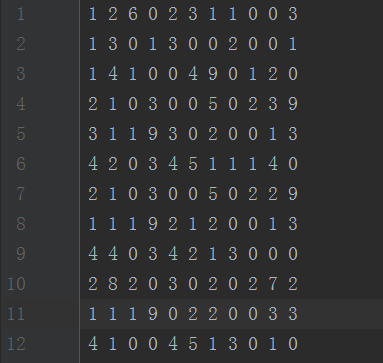
### 3.3 示例

代码如下：

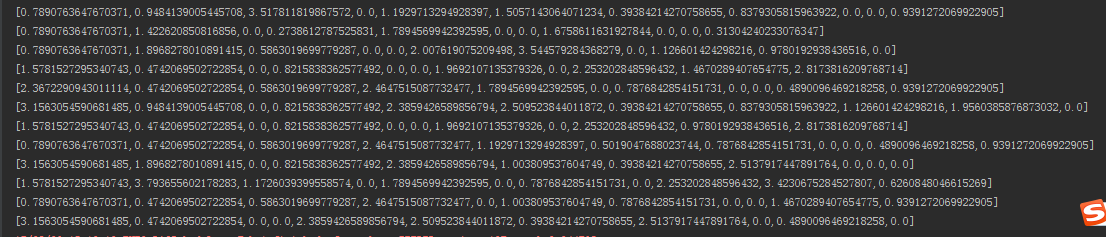




原始数据如下：



数据转换结果如下：



## 4 Normalizer

### 4.1 Normalizer介绍

正则化的过程是将每个样本缩放到单位范数（每个样本的范数为1），spark里面默认范数为2，如果后面要使用如二次型（点积）或者其它核方法计算两个样本之间的相似性这个方法会很有用比如TF-IDF中使用L2正则。

Normalizer主要思想是对每个样本计算其p-范数，然后对该样本中每个元素除以该范数，这样处理的结果是使得每个处理后样本的p-范数（l1-norm,l2-norm）等于1。

### 4.2 Normalizer使用

（1） 首先定义Normalizer类的对象

Normalizer norm=new Normalizer(p= Double.PositiveInfinity);

Double.PositiveInfinity的含义是使用向量里面绝对值最大的那个数作为这的p值

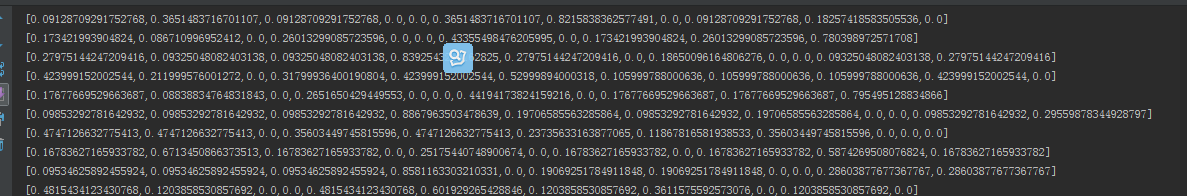
（2） 其次调用该类的transform方法 参数可以是RDD[Vector]也可以是JavaRDD[Vector]也可以是Vector。

### 4.3 示例

依旧使用上面例子的数据，并且数据data是JavaRDD[Vector]类型，处理代码如下：

Normalizer norm=new Normalizer();  
JavaRDD<Vector> normData = norm.transform(data);  
norm.t  
normData.foreach(new VoidFunction<Vector>() {  
 @Override  
 public void call(Vector vector) throws Exception {  
 System.*out*.println(vector);  
 }  
});

正则化结果为如下图：



## 5 Feature selection

### 5.1 Feature selection介绍

通过卡方的方法进行特征的选择，选取与标签列最有用的特征，然后进行训练

### 5.2 Feature selection使用

（1） 首先创建卡方选择器ChiSqSelector

（2） 其次调用ChiSqSelector的fit方法参数为RDD[Vector],如果是JavaRDD需要转一下

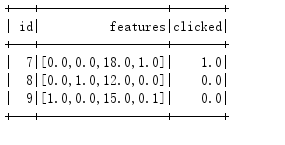
（3） fit方法会返回ChiSqSelectorModel训练的模型，然后通过调用transform方法可以进选择有价值的列出来，参数为JavaRDD<Vector> RDD<Vector> Vector都可以

### 5.3 示例

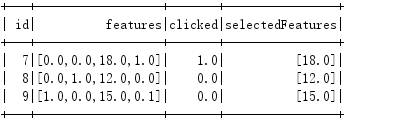
代码如下：

JavaRDD<LabeledPoint> points = sc.textFile("data/sample\_lda\_data.txt", 1).map(  
 new Function<String, LabeledPoint>() {  
 @Override  
 public LabeledPoint call(String s) throws Exception {  
 String[] ss = s.split(" ");  
 double[] d = new double[ss.length - 1];  
 for (int i = 1; i < ss.length; i++) {  
 d[i - 1] = Double.*valueOf*(ss[i]);  
 }  
 return new LabeledPoint(Double.*valueOf*(ss[0]), Vectors.*dense*(d));  
 }  
 }  
);  
JavaRDD<LabeledPoint> discretizedData = points.map(  
 new Function<LabeledPoint, LabeledPoint>() {  
 @Override  
 public LabeledPoint call(LabeledPoint lp) {  
 final double[] discretizedFeatures = new double[lp.features().size()];  
 for (int i = 0; i < lp.features().size(); ++i) {  
 discretizedFeatures[i] = Math.*floor*(lp.features().apply(i) / 16);  
 }  
 return new LabeledPoint(lp.label(), Vectors.*dense*(discretizedFeatures));  
 }  
 });  
  
  
ChiSqSelector selector = new ChiSqSelector(2);  
  
final ChiSqSelectorModel transformer = selector.fit(discretizedData.rdd())  
JavaRDD<LabeledPoint> filteredData = discretizedData.map(  
 new Function<LabeledPoint, LabeledPoint>() {  
 @Override  
 public LabeledPoint call(LabeledPoint lp) {  
 return new LabeledPoint(lp.label(), transformer.transform(lp.features()));  
 }  
 }  
);

例如输入的数据集如下，clicked是标签列 features是特征向量：



输出结果如下selectedFeatures是选择出来的特征列 下面是选择1个特征出来的效果图：



## 6 PCA

### 6.1 PCA介绍

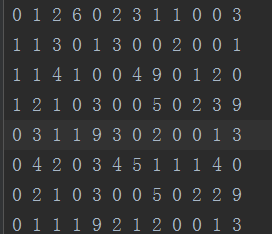
PCA（Principal Component Analysis）是一种常用的数据分析方法。PCA通过线性变换将原始数据变换为一组各维度线性无关的表示，可用于提取数据的主要特征分量，常用于高维数据的降维。

### 6.2 PCA使用

* 1. 首先创建PCA类对象PCA p=new PCA(2); 参数2代表主成分的个数也就是我们降维要降低到的维度；
  2. 其次，调用PCA类的fit方法参数可以为JavaRDD<Vector> RDD<Vector>；
  3. 最后fit方法返回PCAModel然后我们可以调用transform方法，参数为JavaRDD<Vector> RDD<Vector> Vector都可以，用这个模型进行相应的数据处理。

### 6.3 示例

输入数据如下，第一列为label 其他类为features需要封装成Vector

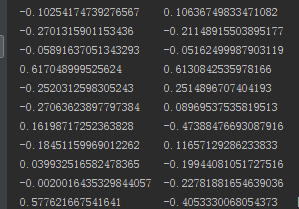


接着我们对数据进行PCA降维处理 设置k为2，每一列代表一个主成分，每一行代表一个特征，上面的数据12列 （11个特征列 1个标签列），

因此输出结果里面有11行 每一行代表一个特征 然后两列是该特征的主成分，

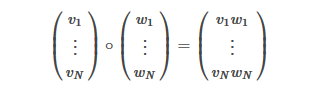
如果想要转换数据我们可以用原始数据12\*11的矩阵\*11\*2的矩阵得到12\*2的矩阵从而实现了 数据的降维度。

输出图如下;



## 7 ElementwiseProduct

### 7.1 ElementwiseProduct介绍



该类的功能是将一个向量乘以另个一向量得到新的向量

### 7.2 ElementwiseProduct使用

* 1. 首先构建初始数据RDD<Vector> 或者JavaRDD<Vector>
  2. 其次创建一个转换向量Vector transformingVector = Vectors.dense(0.0, 1.0, 2.0);
  3. 然后构建ElementwiseProduct 类并且把转换向量传入

ElementwiseProduct transformer = new ElementwiseProduct(transformingVector);

* 1. 然后调用ElementwiseProduct类的transform方法参数可以是RDD<Vector> 或者JavaRDD<Vector> 或者Vector

### 7.3 示例

简单代码如下：

JavaRDD<Vector> data = sc.parallelize(Arrays.*asList*(  
 Vectors.*dense*(1.0, 2.0, 3.0), Vectors.*dense*(4.0, 5.0, 6.0)));  
Vector transformingVector = Vectors.*dense*(0.0, 1.0, 2.0);  
ElementwiseProduct transformer = new ElementwiseProduct(transformingVector);  
JavaRDD<Vector> transformedData = transformer.transform(data);

输出结果如下：

