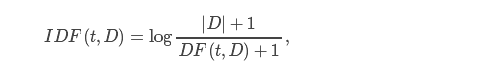
* [TF-IDF](http://spark.apache.org/docs/latest/mllib-feature-extraction.html" \l "tf-idf)
* [Word2Vec](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#word2vec)
  + [Model](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#model)
  + [Example](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#example)
* [StandardScaler](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#standardscaler)
  + [Model Fitting](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#model-fitting)
  + [Example](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#example-1)
* [Normalizer](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#normalizer)
  + [Example](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#example-2)
* [ChiSqSelector](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#chisqselector)
  + [Model Fitting](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#model-fitting-1)
  + [Example](http://spark.apache.org/docs/latest/mllib-feature-extraction.html#example-3)

# 1.TF-IDF

词频- 逆文档频率（TF-IDF）是在文本挖掘中广泛使用的特征向量化方法，以反映词语对语料库中的文档的重要性。用t表示一个术语，用d表示文档，用D表示语料库。词语频率TF（t，d）是词语t出现在文档d中的次数，而文档频率DF（t，D）是包含词语t的文档数T。如果我们仅使用词语频率来衡量重要性，则过于强调经常出现但很少提供有关文档的信息的词语，例如“a”，“the”和“of”。 如果词语在语料库中经常出现，则表示它不包含有关特定文档的特殊信息。逆文档频率是词语提供的信息量的数字度量：



其中| D |是语料库中的文档总数。由于使用了对数，如果词语出现在所有文档中，则其IDF值变为0。请注意，应用平滑项以避免语料库外的词语除以零。TF-IDF指标只是TF和IDF的产物：



词语频率和文档频率的定义有几种变体。在spark.mllib中，我们将TF和IDF分开以使它们变得灵活。我们的术语频率实现利用散列技巧(hashing trick)。通过应用散列函数将原始特征映射到索引（术语）。然后，基于映射的指数计算词语频率。这种方法避免了计算全局词语到索引映射的需要，这对于大型语料库来说可能是昂贵的，但是它遭受潜在的散列冲突，其中不同的原始特征可能在散列之后变成相同的词语。为了减少冲突的可能性，我们可以增加目标特征维度，即哈希表的桶的数量。默认的特征维度是220=1048576.

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.Arrays;  **import** java.util.List;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.feature.HashingTF;  **import** org.apache.spark.mllib.feature.IDF;  **import** org.apache.spark.mllib.feature.IDFModel;  **import** org.apache.spark.mllib.linalg.Vector;  **public** **class** TFIDFExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("TFIDFExample").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  JavaRDD<String> data = sc  .textFile("F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\kmeans\_data.txt");  JavaRDD<List<String>> documents = data.map(d -> {  String[] strArr = d.split(" ");  **return** Arrays.*asList*(strArr);  });  HashingTF hashingTF = **new** HashingTF();  JavaRDD<Vector> tf = hashingTF.transform(documents);  tf.foreach(v -> System.***out***.println(v));  System.***out***.println();  tf.cache();  IDFModel model = **new** IDF().fit(tf);  JavaRDD<Vector> tfidf = model.transform(tf);  tfidf.foreach(t -> System.***out***.println(t));  System.***out***.println();  // 出现在不小于两个文档中的词  **new** IDF(2).fit(tf).transform(tf).foreach(x -> System.***out***.println(x));  ;  sc.stop();  }  } |

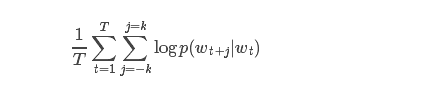
# 2. Word2Vec

Word2Vec计算单词的分布式矢量表示。分布式表示的主要优点是类似的单词在向量空间中很接近，这使得对新颖模式的推广更容易，并且模型估计更加稳健。分布式矢量表示被证明在许多自然语言处理应用中是有用的，例如命名实体识别，消歧，解析，标记和机器翻译。

Word2Vec在文本语料库中创建单词的向量表示。 该算法首先从语料库构造词汇表，然后学习词汇表中单词的向量表示。 矢量表示可以用作自然语言处理和机器学习算法中的特征。

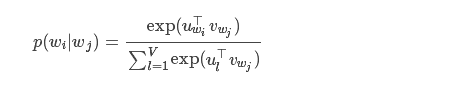
## 2.1模型

在我们的Word2Vec实现中，我们使用了skip-gram模型。skip-gram的训练目标是学习擅长在同一句子中预测其上下文的单词矢量表示。在数学上，给定一系列训练单词w1，w2，...，wT，skip-gram模型的目标是最大化平均对数似然



其中k是训练窗口的大小。

在skip-gram模型中，每个单词w与两个向量uw和vw相关联，这两个向量分别是作为单词w和上下文的向量表示。正确预测给定单词wj单词wi的概率由softmax模型确定，即



其中,V是词汇表大小。具有softmax的skip-gram模型是代价昂贵的，因为计算logp（wi | wj）的成本与V成比例，V可以容易地达到数百万。为了加速Word2Vec的训练，我们使用了分层softmax，它将logp（wi | wj）的计算复杂度降低到O（log（V））

## 2.2 代码示例

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** java.util.Arrays;  **import** java.util.List;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.feature.Word2Vec;  **import** org.apache.spark.mllib.feature.Word2VecModel;  **import** scala.Tuple2;  **public** **class** Word2VecExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("TFIDFExample").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  JavaRDD<String> data = sc  .textFile("F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_lda\_data.txt");  JavaRDD<List<String>> corpus = data.map(s -> Arrays.*asList*(s.split(" ")));  Word2Vec word2Vec = **new** Word2Vec();  Word2VecModel word2VecModel = word2Vec.fit(corpus);  Tuple2<String, Object>[] tuple = word2VecModel.findSynonyms("1", 5);  **for** (**int** i = 0; i < tuple.length; i++) {  System.***out***.println(tuple[i]);  }  sc.stop();  }  } |

# 3. StandardScaler

通过使用训练集中样本的列摘要统计信息缩放到单位方差和/或删除均值来标准化特征。这是一个非常常见的预处理步骤。例如，当所有特征具有单位方差和/或零均值时，支持向量机的RBF内核或L1和L2正则化线性模型通常工作得更好。标准化可以在优化过程中提高收敛速度，并且还可以防止在模型训练期间对具有非常大的差异的特征施加过大的影响。

## 3.1 模型拟合

StandardScaler在构造函数中具有以下参数：

* **withMean：**默认情况下为False。 在缩放之前使用均值将数据居中。 它将构建密集输出，因此在应用稀疏输入时要小心。
* **withStd**:默认为True。 将数据缩放到单位标准偏差。

我们在StandardScaler中提供了一个拟合方法，可以输入RDD [Vector]，学习汇总统计，然后返回一个模型，该模型可以将输入数据集转换为单位标准差和/或零均值特征，具体取决于我们如何配置StandardScaler。

该模型实现了VectorTransformer，它可以在Vector上应用标准化以生成转换后的Vector或在RDD [Vector]上生成转换后的RDD [Vector]。

## 3.2 代码示例

下面的示例演示了如何以libsvm格式加载数据集，并标准化特征，以便新特征具有单位标准差和/或零均值。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.feature.StandardScaler;  **import** org.apache.spark.mllib.feature.StandardScalerModel;  **import** org.apache.spark.mllib.linalg.Vector;  **import** org.apache.spark.mllib.linalg.Vectors;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** StandardScalerExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("TFIDFExample").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(sc.sc(), path).toJavaRDD();  StandardScalerModel scaler1 = **new** StandardScaler().fit(data.map(l -> l.features()).rdd());  StandardScalerModel scaler2 = **new** StandardScaler(**true**, **true**).fit(data.map(x -> x.features()).rdd());  StandardScalerModel scaler3 = **new** StandardScalerModel(scaler2.std(), scaler2.mean());  // System.out.println(scaler1.mean() + " " + scaler2.mean() + " " + scaler3.mean());  JavaRDD<Tuple2<Double, Vector>> tuple2 = data  .map(x -> **new** Tuple2<>(x.label(), scaler1.transform(x.features())));  tuple2.foreach(t -> System.***out***.println(t));  System.***out***.println();  JavaRDD<Tuple2<Double, Vector>> tuple21 = data  .map(x -> **new** Tuple2<>(x.label(), scaler2.transform(Vectors.*dense*(x.features().toArray()))));  tuple21.foreach(t -> System.***out***.println(t));  }  } |

# 4. Normalizer(正则化)

Normalizer将单个样本缩放为具有单位Lp范数。这是文本分类或聚类的常用操作。例如，两个L2标准化TF-IDF值的点积是向量的余弦相似性。

Normalizer在构造函数中具有以下参数：

* **p** ：Lp空间中的标准化，默认情况下p = 2。

Normalizer实现VectorTransformer，它可以在Vector上应用正则化以生成转换后的Vector或在RDD [Vector]上生成转换后的RDD [Vector]。

请注意，如果输入的范数为零，则它将返回输入向量。

代码示例：

下面的示例演示了如何以libsvm格式加载数据集，并使用L2范数和L∞范数对特征进行规范化。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.mllib.feature.Normalizer;  **import** org.apache.spark.mllib.linalg.Vector;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.util.MLUtils;  **import** scala.Tuple2;  **public** **class** NormalizerExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("NormalizerExample").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(sc.sc(), path).toJavaRDD();  Normalizer normalizer1 = **new** Normalizer();  Normalizer normalizer2 = **new** Normalizer(Double.***POSITIVE\_INFINITY***);  JavaRDD<Tuple2<Double, Vector>> data1 = data  .map(x -> **new** Tuple2<>(x.label(), normalizer1.transform(x.features())));  data1.foreach(x -> System.***out***.println(x));  System.***out***.println();  JavaRDD<Tuple2<Double, Vector>> data2 = data  .map(x -> **new** Tuple2<>(x.label(), normalizer2.transform(x.features())));  data2.foreach(x -> System.***out***.println(x));  }  } |

# 5. ChiSqSelector

特征选择试图识别用于模型构造的相关特征。它减小了特征空间的大小，这可以提高速度和统计学习行为。ChiSqSelector实现了Chi-Squared特征选择。它使用具有分类特征的标记数据进行操作。ChiSqSelector使用Chi-Squared独立性测试来决定选择哪些特征。它支持五种选择方法：num Features，percentile，for，fdr，fwe：

* numTopFeatures根据卡方检验选择固定数量的顶部特征。 这类似于产生具有最强预测能力的特征。
* percentile :与numTopFeatures类似，但选择所有特征的一小部分而不是固定数字。
* **fpr**:选择p值低于阈值的所有特征，从而控制fp(false positive)的选择比率。
* **fdr**:使用Benjamini-Hochberg过程来选择错误发现率低于阈值的所有特征。
* **few**:选择p值低于阈值的所有特征。 阈值按1 / numFeatures缩放，从而控制选择的family-wise错误率。

默认情况下，选择方法为numTopFeatures，默认顶部要素数设置为50.用户可以使用setSelectorType选择选择特征的方法。可以使用保留的验证集来调整要选择的特征的数量。

## 5.1 模型拟合

fit方法接受具有分类特征的RDD [LabeledPoint]输入，学习摘要统计，然后返回ChiSqSelectorModel，它可以将输入数据集转换为缩小的特征空间。ChiSqSelectorModel可以应用于Vector以生成缩减的Vector，也可以应用于RDD [Vector]以生成缩减的RDD [Vector]。注意，用户还可以通过提供所选特征索引的数组（必须按升序排序）来手动构造ChiSqSelectorModel。

## 5.2 代码示例

以下示例显示了ChiSqSelector的基本用法。 使用的数据集具有由灰度值组成的特征矩阵，每个特征的灰度值从0到255不等。

|  |
| --- |
| **package** com.cb.spark.mllib;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **import** org.apache.spark.api.java.function.Function;  **import** org.apache.spark.mllib.feature.ChiSqSelector;  **import** org.apache.spark.mllib.feature.ChiSqSelectorModel;  **import** org.apache.spark.mllib.linalg.Vector;  **import** org.apache.spark.mllib.linalg.Vectors;  **import** org.apache.spark.mllib.regression.LabeledPoint;  **import** org.apache.spark.mllib.util.MLUtils;  **public** **class** ChiSqSelectorExample {  **public** **static** **void** main(String[] args) {  SparkConf conf = **new** SparkConf().setAppName("NormalizerExample").setMaster("local");  JavaSparkContext sc = **new** JavaSparkContext(conf);  String path = "F:\\Learning\\java\\project\\LearningSpark\\src\\main\\resources\\sample\_libsvm\_data.txt";  JavaRDD<LabeledPoint> data = MLUtils.*loadLibSVMFile*(sc.sc(), path).toJavaRDD();  data = data.map(**new** Function<LabeledPoint, LabeledPoint>() {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** LabeledPoint call(LabeledPoint v1) **throws** Exception {  **double** label = v1.label();  Vector v = v1.features();  **double**[] d = **new** **double**[v.size()];  **for** (**int** i = 0; i < v.size(); i++) {  **double** newValue = Math.*floor*(v.apply(i) / 16);  d[i] = newValue;  }  Vector newV = Vectors.*dense*(d);  **return** **new** LabeledPoint(label, newV);  }  });  ChiSqSelector selector = **new** ChiSqSelector(50);  ChiSqSelectorModel transformer = selector.fit(data.rdd());  JavaRDD<LabeledPoint> filteredData = data  .map(lp -> **new** LabeledPoint(lp.label(), transformer.transform(lp.features())));  filteredData.foreach(l -> System.***out***.println(l));  }  } |